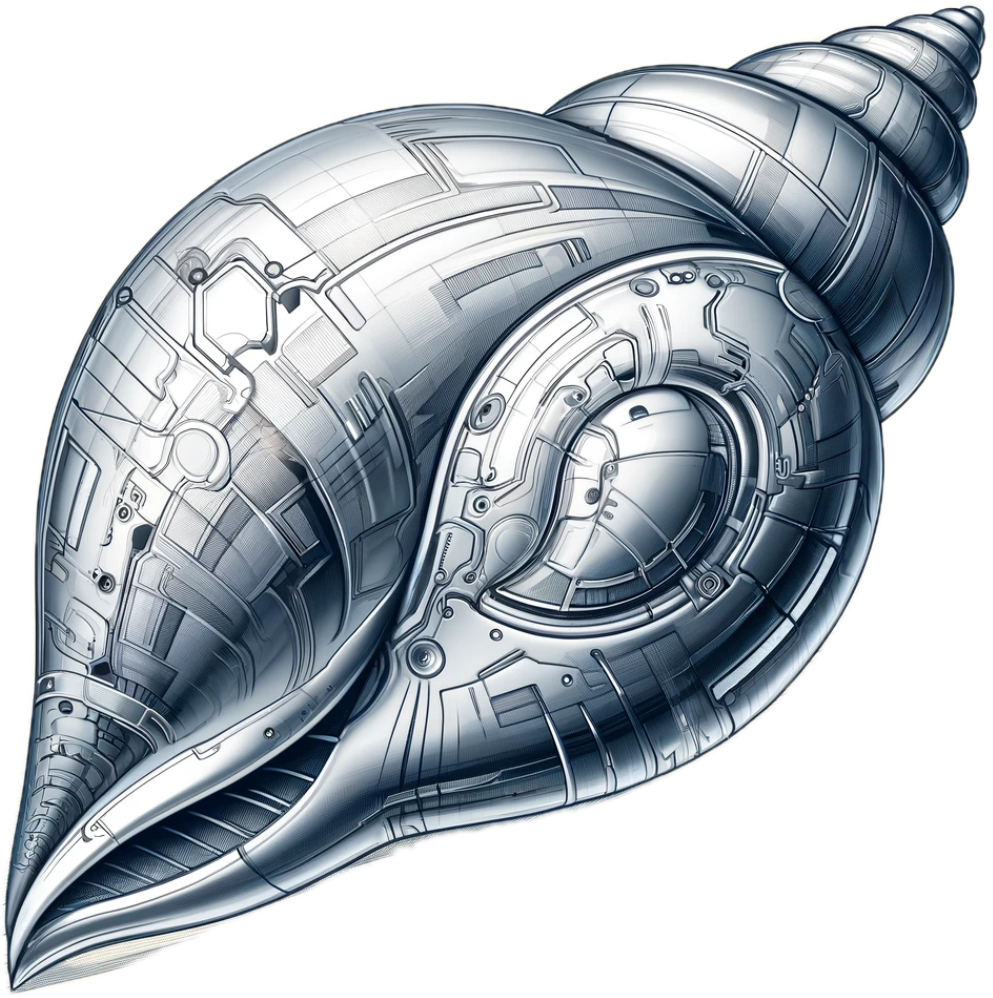

[10]

A Strong Typed Computational Model for Shell Languages

Andersen, Christoffer Lind
clan19@student.aau.dk

Bonderup, Nikolai Aaen
nbonde19@student.aau.dk

June 14, 2024

Derpartment of Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY

STUDENT REPORT

Title:
A Strong Typed Computational Model for Shell Languages

Theme:
Distributed systems - Programming languages

Project Period:
January 2024 - June 2024

Project Group:
Fri-91211-1

Participant(s):
Nikolai Aaen Bonderup
Christoffer Lind Andersen

Supervisor(s):
Hans Hüttel

Copies: 1

Page Numbers: 82

Date of Completion:
June 14, 2024

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Resume

I denne afhandling præsenterer vi λ_{sh}, en ny tilgang til implementering af et stærkt typesystem i et shell-programmeringssprog, hvor strenge spiller en central rolle. Traditionelle shell sprog, som Bash og PowerShell, er kraftfulde værktøjer til programmering, men lider under svage typesystemer, der kan fører til køretids-fejl, vedligeholdelsesproblemer og sikkerhedssårbarheder. Vores arbejde med λ_{sh} adresserer disse problemer ved at udnytte principperne fra λ-kalkulen og formel typeteori til at skabe et robust, statisk-typet shell-sprog.

De vigtigste bidrag fra denne afhandling inkluderer:

1. **Design og semantik af λ_{sh}:** Vi har defineret syntaks, operationel semantik og typesystem for λ_{sh}, hvilket sikrer, at sproget understøtter forudsigelige og konsistente strengoperationer, samtidig med at typesikkerhed opretholdes.

2. **Formel analyse:** Gennem formelle beviser har vi demonstreret typesystemets korrekthed, hvilket sikrer, at korrekt typede programmer ikke støder på typerelaterede køretids-fejl.

3. **Grundlag for fremtidige udvidelser:** Vi har fokuseret på at etablere de grundlæggende aspekter af λ_{sh}, som kan udvides med mere avancerede funktioner såsom records, standard bibliotek, osv.

Ved at kombinere fleksibiliteten og nytten af traditionelle shell-miljøer med robustheden af et stærkt typesystem, tilbyder λ_{sh} et kraftfuldt specifikation til udviklere, der ønsker at skrive mere pålidelige, vedligeholdelsesvenlige og sikre shell-scripts.

Fremtidigt arbejde Selvom denne afhandling lægger et omfattende grundlag, er der flere retninger for fremtidig udvikling af λ_{sh}:

1. **Udvidelse af semantik:** Der er behov for yderligere arbejde med at modellere shell-miljøet mere fuldstændigt, herunder avancerede filsysteminteraktioner, processtyring og håndtering af input/output.

2. **Avancerede sprogfunktioner:** Fremtidige udvidelser kunne inkorporere mere komplekse datastrukturer, modulære programmeringsmuligheder og fejlbehandlingsmekanismer.

3. **Praktisk implementering:** En faktisk implementering af λ_{sh}, inklusive en fortolker og et standardbibliotek, er afgørende for at validere de teoretiske koncepter og gøre λ_{sh} til et praktisk værktøj til anvendelse i den virkelige verden.

Arbejdet præsenteret i denne afhandling adresserer væsentlige svagheder i traditionelle shell-scriptsprog og lægger grunden til udviklingen af mere pålidelige og sikre shell-programmeringsværktøjer. Ved at bygge videre på de principper og fundamenter, der er etableret her, har λ_{sh} potentialet til at have en indvirkning på området for shell-scripting og automatisering, og tilbyde udviklere et mere kraftfuldt og pålideligt sprog til deres shell scripting-behov.

Abstract

Shell programming languages, such as Bash and PowerShell, are widely utilised for scripting and automation in various computing environments. However, these languages often suffer from weak typing or non-existent type systems, leading to runtime errors, maintenance challenges, and security vulnerabilities. This thesis introduces λ_{sh}, a novel functional shell programming language designed to address the inherent weaknesses of traditional shell scripting languages by incorporating a strong, statically-typed system.

λ_{sh} is based on the λ-calculus, leveraging its theoretical foundations to create a robust and reliable shell scripting environment. The language integrates explicit constructs for string manipulation, predictable and consistent handling of string operations, and a formal model for the shell environment. By introducing a strong type system, λ_{sh} ensures type safety, early error detection, and improved script readability and maintainability.

This thesis provides a detailed analysis of the theoretical foundations of λ_{sh}, including lambda calculus, formal semantics, and type theory. It discusses the design principles, design considerations, and formal operational semantics of the language. Furthermore, the thesis presents a comprehensive exploration of string concatenation and interpolation, demonstrating how λ_{sh} balances the flexibility of weak typing with the safety of strong typing.

Acknowledgements

We would like to express our sincere gratitude to our advisor, Hans Hüttel, for his invaluable advice and support in the creation of this thesis. His guidance and expertise have been instrumental in our work, research, and writing process, and we are deeply appreciative of his time and effort.

Contents

Chapter 1

Introduction

1.1 The nature of shell programming

Shell languages, such as Bash, PowerShell, and others commonly used for scripting in Unix and Windows environments, are characterised by their non-existent or very weak typing systems. This means that variables in these languages are not bound to a specific type, and the type of data a variable can hold may change during the execution of a script. The flexibility afforded by weak typing is particularly evident in the ubiquitous use of strings as the primary data type.

In shell languages, almost all data is treated as a string[3]. This design choice simplifies many aspects of scripting, especially given the text-based nature of command-line interfaces. Commands, filepaths, input/output operations, and numerical values are manipulated as strings. This string-centric paradigm allows for a highly flexible and dynamic scripting environment where variables can be effortlessly concatenated, split, or parsed without the need for explicit type declarations, since everything is considered the same type.

For instance, in a typical Bash script, one might see variables assigned and manipulated without regard to their underlying data types:

```
#!/bin/bash
a="123"
b="456"
result=$((a + b))
echo "The sum is: $result"
```

In this example, `a` and `b` are initially treated as strings but are implicitly converted to integers for the arithmetic operation. The result is then seamlessly converted back to a string for output. This implicit type conversion is an important mechanic in weakly typed languages, reducing the overhead of managing data types but also introducing potential pitfalls if not carefully managed.

The implicit conversion between types is both a strength and a potential source of bugs in shell scripting. While it allows for rapid development and reduces boilerplate code, it can lead to unexpected behaviour if the script inadvertently operates on data in an unintended type[6]. For example, attempting to perform arithmetic on non-numeric strings will result in errors or incorrect results, which might not be immediately apparent:

```
#!/bin/bash
a="123abc"
b="456"
result=$((a + b))  # This will cause an error
```

Here, `a` contains non-numeric characters, leading to an error when an arithmetic operation is attempted. Such issues highlight the trade-offs inherent in the weakly typed nature of shell languages. These types of errors cannot be caught before runtime, because of how the implicit conversion works.

The predominance of strings and the weak typing system in shell languages are pragmatic choices that cater to the needs of shell scripting. These uses often require scripts that can handle a range of input types, interface with numerous command-line tools, and perform text processing tasks efficiently. The advantage of a weak typing system and the versatile use of strings is that it enables quick prototyping and straightforward manipulation of textual data, an important task in the domain, therefore making shell languages a powerful tool for automation and scripting in diverse environments.

However, this flexibility necessitates a disciplined approach to scripting. Developers must be vigilant about input validation, type checking, and error handling to mitigate the risks associated with implicit type conversions and the predominance of strings, and these tasks cannot be automated by tools because of the language design.

1.2 Moving towards stronger type systems

Despite the benefits of weak typing and the extensive use of strings, we have an interest in incorporating stronger type systems into shell languages. A stronger type system would enforce more rigorous type checking at compile time or runtime, reducing the likelihood of type-related errors and improving overall script reliability and maintainability as discussed in our previous report "Requirements for a functional shell programming language"[2].

The advantages of a stronger type system are plentiful. Firstly, with explicit type declarations and enforcement, many common scripting errors could be caught early, reducing debugging time and improving script stability. Furthermore, scripts with clear type annotations are easier to read and understand, making maintenance and collaboration more straightforward. And lastly, certain optimisations become possible when the interpreter or compiler knows the exact types of variables in advance, potentially leading to more efficient execution.

There are several ways to go about implementing a stronger type system in a shell language. Firstly, we could introduce optional type annotations that allow developers to specify the types of variables and function parameters while retaining backward compatibility with untyped scripts. We could also implement type inference mechanisms that automatically deduce the types of variables based on their usage, providing the benefits of strong typing without the verbosity of explicit type annotations. The issue here is that we know that there are cases where such inference is not possible. There is also the possiblity of implementing an entirely new language that is strongly typed, but has mechanisms for easily working with the string representation of values. In a sense, we want a type system that combine the flexibility of weak typing with the robustness of strong typing. This approach could allow variables to be explicitly typed where necessary while permitting implicit conversions for simpler, more dynamic tasks.

Moreover, we are particularly interested in developing type systems that can be formally proven to be sound. A sound type system guarantees that if a program type checks successfully, it will not produce type errors during execution. This property is critical for ensuring the reliability and robustness of scripts, particularly in environments where script failures can have significant consequences.

By moving towards stronger and provably sound type systems, shell languages can enhance their robustness and maintain their ease of use, providing developers with powerful tools for reliable and efficient scripting. This direction not only improves the scripting experience but also opens up new possibilities for safer and more maintainable automation solutions with shell scripts[2].

1.3 The λ_{sh} language

To address these issues, we have created the λ_{sh} language, a formal definition of a functional shell programming language that solves the mentioned problems by incorporating several key features.

Firstly, λ_{sh} provides explicit constructs for string manipulation, eliminating ambiguities associated with word-expansion, word-splitting, and string interpolation. This ensures that string operations are predictable and consistent, reducing the risk of subtle bugs that arise from implicit string manipulations.

Secondly, the language includes a model of the shell environment.

The most important contribution of λ_{sh} is to show that we can have a strong type system in a shell language while maintaining much of the flexibility of traditional shell languages. To overcome the limitations of typical shells weak and nonexistent type system, λ_{sh} incorporates a strong, statically typed system. This type system catches type errors at compile time, ensuring that variables and functions are used correctly according to their types. This reduces the likelihood of runtime errors and makes scripts easier to understand and maintain. The key to solving this problem is to introduce at type system where strings play a central role.

While λ_{sh} benefits from a strong type system, it also allows for subtype polymorphism where appropriate. This is managed through a predefined set of subtype mapping, ensuring that operations are still safe and well-defined.

λ_{sh} is based on the λ-calculus, a model of computation that serves as the foundation for many functional programming languages. The language draws inspiration from Haskell, SML, and Lisp, adopting a declarative programming style that emphasizes immutability and the composition of functions. This paradigm is well-suited for the shell scripting domain, where composing commands is a common practice.

λ_{sh} is defined by a formal semantics that specify the behaviour of its constructs precisely. The type system is designed to be provably sound, ensuring that well-typed programs do not produce type errors during execution. This is achieved through formal proofs based on established type theory principles, guaranteeing the reliability and robustness of scripts.

Chapter 2

Theoretical foundation

2.1 Introduction

The development of λ_{sh} is grounded in a robust theoretical foundation that draws upon concepts from λ-calculus, formal semantics, and type theory. This section delves into the core theoretical principles that underpin the language, providing a rigorous basis for its design and implementation.

Understanding the theoretical underpinnings of λ_{sh} is essential to understand the design and the guarantees promised by the language.

Firstly, we discuss formal semantics, which offer a precise description of how λ_{sh} programs are executed. Here we describe the formalism we use to describe the language. By defining the operational semantics, we specify the rules that govern the execution of commands and expressions within the language. This formalism ensures that the behaviour of λ_{sh} programs is well-defined and predictable.

Next, we examine type theory, which categorises expressions into types and enforces rules for type correctness. The type system of λ_{sh} ensures that operations are performed on compatible types, preventing type errors and enhancing the reliability of programs. We will explore how the type system is defined, the rules for type checking, and the properties that ensure type safety.

By establishing a solid theoretical foundation, we aim to provide a comprehensive understanding of the principles that guide the design of λ_{sh}. This foundation not only supports the implementation of the language but also facilitates reasoning about its behaviour and properties, ensuring that λ_{sh} is both powerful and reliable.

2.2 Formal operational semantics

Formal operational semantics is a rigorous mathematical framework used to describe the behaviour of programs within a programming language. It provides a precise and unambiguous way to define how program statements and expressions are executed[8]. Syntax defines the structure of valid programs in the language using formation rules and an operational semantics use that syntax to define transition systems based on configurations.

The benefits of formal operational semantics can be summarised as the following:

- **Precision**: Eliminates ambiguity in language specifications that you get when a language is specified informally.
- **Verification**: Facilitates formal verification and reasoning about program correctness.
- **Language Design**: Aids in the design and implementation of new programming languages by providing a clear specification of expected behaviours.

2.2.1 Configurations

Configurations represent the state of a program at any point during its execution. This includes:

- The current state of the memory (variable bindings, heap, etc.).
- The current point of execution (control stack, program counter, etc.).

In an operational semantics it is typically a bit more abstract than looking at control stack and program counter, but the general idea holds [8].

2.2.2 Transitions

Transitions are rules that describe how a program moves from one configuration to another. How transitions are specified depends on the kind of operational semantics one is creating. Fundamentally, there are two kinds: big-step semantics and small-step semantics. Small-step semantics breaks down program execution into individual steps. Each step represents a single computational action, such as evaluating an expression or executing a single statement. It provides a detailed trace of execution in an atomic manner. Big-step semantics describes the overall result of executing a complete program or large blocks of code. It focuses on the final outcome rather than the individual steps. Transition rules are defined as $(C, S) \to (C', S')$, meaning configuration C in state S transitions to configuration C' in state S'. Evaluation is typically written as $\langle C, S\rangle \to S'$, meaning that configuration C in state S evaluates to state S' [8].

Transition rules are used to define transitions and evaluations formally. These are often written in the form of logical rules. Rules are written on the form

$$\frac{premises}{conclusion}$$

The premises must hold in order to use the rule, as an example we have a big-step rule for the plus operator:

$$\frac{\langle E_1, S\rangle \to \langle v_1, S'\rangle \quad \langle E_2, S'\rangle \to \langle v_2, S''\rangle}{\langle E_1 + E_2, S\rangle \to \langle v_1 + v_2, S''\rangle}$$

This rule states that to evaluate $E_1 + E_2$ in state S, first evaluate E_1 to get v_1 and new state S', then evaluate E_2 to get v_2 and new state S'', and finally compute the result $v_1 + v_2$ in state S''.

2.2.3 Example of language specification

In order to demonstrate how a big-step and a small-step semantics are formed, we need an example.

Consider the simple language with arithmetic expressions and assignment as statements in Figure 2.1. First, we define a syntax for each syntactic category of the language (expressions and statements)

$$E ::= n \mid x \mid E_1 + E_2$$
$$S ::= x := E \mid S_1; S_2$$

Figure 2.1: Example formation rules.

For the simple abstract syntax in Figure 2.1, we will show an example small-step and big-step semantics.

State

As part of both example semantics we introduce the state δ, which is a mapping from variables to values. We use this to store the contents of variables. An example of a state could be $\{x \mapsto 3, y \mapsto 6\}$.

Small-step semantics

Again, small-step semantics describes the execution of a program as a series of individual steps. The following is a definition of the transition rules as a small-step semantic, where δ is the state (a mapping from variables to values). Note that the transitions of this small language is of the form $\langle e, \delta\rangle \to \langle e', \delta'\rangle$

We define a rule for each syntactic construct in the language, additional rules may be needed to evaluate every step of the computation. First, we consider the rules for expressions (E) in Figure 2.2.

$$\text{[CONST]} \quad \frac{}{\langle n, \delta\rangle \to \langle n, \delta\rangle}$$

$$\text{[VAR]} \quad \frac{\delta(x) = n}{\langle x, \delta\rangle \to \langle n, \delta\rangle}$$

$$\text{[PLUS-1]} \quad \frac{\langle E_1, \delta\rangle \to \langle E_1', \delta'\rangle}{\langle E_1 + E_2, \delta\rangle \to \langle E_1' + E_2, \delta'\rangle}$$

$$\text{[PLUS-2]} \quad \frac{\langle E_2, \delta\rangle \to \langle E_2', \delta'\rangle}{\langle v_1 + E_2, \delta\rangle \to \langle v_1 + E_2', \delta'\rangle}$$

$$\text{[PLUS-3]} \quad \frac{v_3 = v_1 + v_2}{\langle v_1 + v_2, \delta\rangle \to \langle v_3, \delta\rangle}$$

Figure 2.2: Example small-step semantics for expressions.

Now, we also have to consider the category of statements, which we do in Figure 2.3.

$$\text{[ASSIGN-1]} \quad \frac{\langle E, \delta\rangle \to \langle E', \delta'\rangle}{\langle x := E, \delta\rangle \to \langle x := E', \delta'\rangle}$$

$$\text{[ASSIGN-2]} \quad \frac{\langle E, \delta\rangle \to \langle E', \delta'\rangle}{\langle x := E, \delta\rangle \to \langle x := E', \delta'\rangle}$$

$$\text{[SEQ-1]} \quad \frac{\langle S_1, \delta\rangle \to \langle S_1', \delta'\rangle}{\langle S_1; S_2, \delta\rangle \to \langle S_1'; S_2, \delta'\rangle}$$

$$\text{[SEQ-2]} \quad \frac{\langle S_2, \delta\rangle \to \langle S_2', \delta'\rangle}{\langle S_1; S_2, \delta\rangle \to \langle v; S_2', \delta'\rangle}$$

$$\text{[SEQ-3]} \quad \frac{}{\langle v; S_2, \delta\rangle \to \langle v, \delta'\rangle}$$

Figure 2.3: Example small-step semantics for statements.

Combined, the reduction rules defined in Figure 2.2 and 2.3 constitutes a complete small-step semantics for the little language defined in Figure 2.1.

Big-step semantics

Big-step semantics describes the execution of a program as a single large step that computes the final result. The form of transitions is the same as the small-step semantics except the rules now evaluate subexpressions completely in on step. Now, we define the big-step semantics for expressions in Figure 2.4.

$$\text{[B-CONST]} \quad \frac{}{\langle n, \delta \rangle \to n}$$

$$\text{[B-VAR]} \quad \frac{\delta(x) = n}{\langle x, \delta \rangle \to n}$$

$$\text{[B-PLUS]} \quad \frac{\langle E_1, \delta \rangle \to v_1 \quad \langle E_2, \delta \rangle \to v_2 \quad v = v_1 + v_2}{\langle E_1 + E_2, \delta \rangle \to v}$$

Figure 2.4: Example big-step semantics for expressions.

We also need a definition of the big-step semantics for statements. These can be seen in Figure 2.5.

$$\text{[B-ASSIGN]} \quad \frac{\langle E, \delta \rangle \to v}{\langle x := E, \delta \rangle \to \delta[x \mapsto v]}$$

$$\text{[B-SEQ]} \quad \frac{\langle S_1, \delta \rangle \to \delta' \quad \langle S_2, \delta' \rangle \to \delta''}{\langle S_1; S_2, \delta \rangle \to \delta''}$$

Figure 2.5: Example big-step semantics for statements.

Combined, the rules in Figure 2.4 and Figure 2.5 makes a complete big-step semantics for the small language defined in Figure 2.1.

Example derivation

Now, let us see a semantic in action. Consider the evaluation of the following program:

$$x := 1; y := x + 2$$

First, we can do the derivation using the small-step semantics.

1. Initial state: $\delta = \{\}$
2. After evaluating $x := 1$:

$$\langle x := 1, \delta \rangle \to \langle \text{skip}, \delta[x \mapsto 1] \rangle = \{x \mapsto 1\}$$

3. After evaluating $y := x + 2$:

$$\begin{aligned} \langle y := x + 2, \{x \mapsto 1\} \rangle &\to \langle y := 1 + 2, \{x \mapsto 1\} \rangle \\ \langle y := 1 + 2, \{x \mapsto 1\} \rangle &\to \langle y := 3, \{x \mapsto 1\} \rangle \\ \langle y := 3, \{x \mapsto 1\} \rangle &\to \langle \text{skip}, \{x \mapsto 1, y \mapsto 3\} \rangle \end{aligned}$$

And the final state after evaluating is: $\{x \mapsto 1, y \mapsto 3\}$. Similarly, we can also do the derivation using the big-step semantic rules.

1. Initial state: $\delta = \{\}$
2. Evaluating $x := 1$:
$$\langle x := 1, \delta \rangle \to \delta[x \mapsto 1] = \{x \mapsto 1\}$$
3. Evaluating $y := x + 2$:
$$\langle x + 2, \{x \mapsto 1\} \rangle \to 3$$
$$\langle y := x + 2, \{x \mapsto 1\} \rangle \to \{x \mapsto 1, y \mapsto 3\}$$

And the final state is: $\{x \mapsto 1, y \mapsto 3\}$

In summary, formal operational semantics provides a framework for understanding and reasoning about program execution through a set of mathematically precise rules. It helps in defining and analysing the behaviour of programming languages in a rigorous way.

2.3 Type systems

A type system is a formal framework used to define and enforce the classification of expressions in a programming language. It ensures that programs adhere to certain rules that prevent type errors, making the programs more predictable, safer, and easier to maintain. Type systems are crucial in catching errors at compile time rather than at runtime, thereby enhancing the reliability and robustness of software[8].

2.3.1 Typing contexts

A typing context, often denoted by the symbol Γ, is a fundamental component of a type system in programming languages. It serves as a formal environment that maintains the associations between variables and their respective types. The typing context plays a role in type checking and type inference, providing the necessary information to determine the types of expressions within a given scope.

In more detail, a type context Γ can be described as follows:

- **Definition:** A type context Γ is a finite mapping from variables to types. It records the type assumptions for the variables that are currently in scope:
$$\Gamma : Variables \rightharpoonup Types$$
- **Notation:** The type context is usually written as a sequence of variable-type pairs:
$$\Gamma = [x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n]$$
where each x_i is a variable and τ_i is the type associated with x_i.
- **Purpose:** The primary purpose of the type context is to provide a framework for type judgments. When checking the type of an expression, the type context is used to look up the types of free variables within the expression. This ensures that the expression is consistent with the type assumptions in the context.
- **Example:** Consider the expression $(\lambda x : \tau.e)$. When determining the type of this expression, the type context Γ is extended with the new binding $x : \tau$. This extended context is then used to type check the body of the lambda abstraction e.

The role of the type context in type judgements can be formally expressed as:

$$\Gamma[x \mapsto \tau] \vdash e : \tau'$$

This notation indicates that under the type context Γ extended with the binding $x : \tau$, the expression e has the type τ'.

In summary, a type context Γ is a structured environment that records the types of variables in scope. It provides the foundation for making accurate type judgements, ensuring that expressions are type-checked in a coherent and deterministic manner.

2.3.2 Type judgements

A type judgement is a formal assertion within the context of a type system that associates an expression with a type under a given typing context. It serves as the foundation for verifying and ensuring that expressions in a programming language adhere to the specified type rules, thus preventing type errors during program execution.

In its most general form, a type judgement is written as:

$$\Gamma \vdash e : \tau$$

where:

- Γ (the typing context) is a context that maps variables to their corresponding types. It represents the assumptions about the types of free variables in the expression.
- e (the expression) is the syntactic entity whose type is being determined.
- τ (the type) is the type assigned to the expression e under the assumptions in Γ.

The type judgment $\Gamma \vdash e : \tau$ reads as "*under the typing context* Γ*, the expression* e *has the type* τ." This relationship is derived using a set of typing rules defined by the type system, which dictate how types are assigned to various constructs in the language. Type judgments play a role in type checking, where they are used to ensure that programs conform to their specified types. This process involves systematically applying the typing rules to derive type judgements for all expressions in the program. If all expressions can be assigned types according to the rules, the program is considered well-typed, indicating that it adheres to the language's type constraints and is free from certain classes of errors.

In summary, a type judgement provides a formal mechanism for associating expressions with types within a type system. It encapsulates the rules and assumptions under which an expression is deemed to have a particular type, thereby contributing to the correctness and reliability of programs.

2.3.3 Subtyping

Subtyping is a fundamental concept in type theory and programming languages, which expresses that one type (the *subtype*) is a specialized version of another type (the *supertype*). This relationship facilitates flexible and reusable code if implemented correctly since functions or data structures designed to work with a supertype can also operate with any of its subtypes[11].

Fundemental to subtyping is the subtyping relation. If type S is a subtype of type T (denoted $S <: T$), an instance of S can be used wherever an instance of T is expected. Futhermore, it is genreally accepted that it should follow the Liskov Substituion Principle, which says that objects of a superclass should be replaceable with objects of a subclass without affecting the program's correctness.

Adding subtyping to a language Subtyping must be designed to maintain type safety. If a subtype can be substituted for a supertype, operations defined for the supertype should remain valid for the subtype. This involves defining type checking rules that incorporate subtyping. This should of course also be reflected in the operational semantics in such a way that we maintain the soundness property. In operational semantics, the semantics describe how the execution of expressions modifies the program state. When subtyping is involved, the semantics must account for an expression of a subtype appearing in a context expecting a supertype, possibly involving implicit or explicit type conversions at runtime. Here, object-oriented languages are a good example of subtyping. In object-oriented languages, subtyping is closely related to inheritance. The operational semantics must specify how methods are dispatched when an object is treated as an instance of a supertype but is actually an instance of a subtype. This involves defining the lookup rules for methods and attributes, ensuring that the correct method (possibly overridden in the subtype) is called. It is of course also related to polymorphism, where a single function can operate on arguments of different types, provided they are subtypes of a common supertype. Again, the operational semantics must handle polymorphic functions correctly, ensuring they operate seamlessly with any subtype passed to them.

Example Consider a simple typed language with subtyping. Suppose we have types `Animal` and `Dog`, with `Dog` being a subtype of `Animal` (`Dog` $<:$ `Animal`).

The subsumption rule allows a `Dog` to be treated as an `Animal`:

$$\text{If } \Gamma \vdash d : \texttt{Dog} \text{ and } \texttt{Dog} <: \texttt{Animal}, \text{ then } \Gamma \vdash d : \texttt{Animal}$$

When the program evaluates an expression involving a `Dog` where an `Animal` is expected, the semantics ensure the correct type handling:

$$(d : \texttt{Dog}) \text{ evaluated in the context of } (a : \texttt{Animal}) \rightarrow \text{treat } d \text{ as } \texttt{Animal}$$

In conclusion, subtyping enhances the flexibility and reusability of code but introduces complexity into the operational semantics of a language.

2.3.4 Soundness of a type system

The soundness of a type system is a fundamental property that ensures the reliability and correctness of a programming language's type-checking mechanism. Essentially, a type system is sound if it guarantees that programs which pass type checking will not encounter certain types of runtime errors. This concept is crucial in providing confidence that type-checked programs adhere to their specified type constraints during execution[11].

Formally, the soundness of a type system can be expressed through two main properties: type safety, which encompasses both progress and preservation.

- **Progress**: This property states that well-typed programs will never get stuck. Specifically, if a program is well-typed, then at any point during its execution, the program is either in a final state (i.e., it has completed execution) or there is a valid step that can be taken to continue execution. In other words, a well-typed program will not reach a state where no further execution steps are possible except for specific, well-defined cases like reaching a value[11].
- **Preservation (subject reduction)**: This property ensures that the types of expressions are maintained throughout execution. Formally, if an expression e has a type T and it takes a step to another expression e' ($e \rightarrow e'$), then the resulting expression e' will also have the type T. This means that the execution of well-typed programs preserves the type correctness of intermediate expressions[11].

Soundness is vital for several reasons:

- **Reliability**: A sound type system ensures that programs behave as expected with respect to their type annotations. This reduces the likelihood of unexpected type-related runtime errors.
- **Optimisation**: Compilers can make optimisations based on the guarantees provided by a sound type system. For example, if the type system guarantees that a variable will always hold an integer, the compiler can generate more efficient machine code.
- **Maintainability**: Type soundness aids in program maintenance by catching type errors at compile time, which makes it easier to refactor and extend codebases without introducing new type-related bugs. This goes in hand with the fact that types provide documentation for a piece of code, allowing for easier understanding of what interfaces different abstractions implement.

We adopt the strategy of syntactic type soundness proposed by Wright and Felleisen[12] which is based on the notion of subject reduction devised by Curry and Feys [4]. if $\Gamma \vdash e : \tau$ and $e \to e'$ then $\Gamma \vdash e' : \tau$ where in $\Gamma \vdash e : \tau$, e is the *subject* and τ is the *predicate*. Subject reduction states that the reduction of the subject(expression) should preserve the predicate(type). To prove soundness start by first showing progress, we do proof by induction on the height of the derivation tree of reduction rules (the reduction rules $\langle e, \delta\rangle \to \langle e', \delta\rangle$) [12]. We start this section by introducing the 2 definitions for the soundness properties and defining what it means to have be a well-typed configuration.

In our soundness proofs in this thesis, we use the following definitions:

Definition 1 (Barendregt's variable convention)**.** *If T_1 ... T_n occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables [7]. The validity of the equivalence relation is unchanged by variable names, assuming that all rules hold, it follows that the names of bound variables are different to the free variables.*

Definition 2 (Preservation/Subject reduction)**.** *We have that if $\Gamma \vdash e : T$ and $e \to e'$, then $\Gamma \vdash e' : T$ for all reduction rules.*

Definition 3 (Progress)**.** *if $\Gamma \vdash e : T$, then e is either a value or there exists an e' such that $e \to e'$.*

Definition 4 (Well-typed configuration)**.** *A configuration is well-typed under the typing environment $\Gamma \vdash \langle e, \delta\rangle : \tau$ if: $\exists \tau.\Gamma \vdash e : \tau$*

When we have shown that progress and preservation holds we have a guarantee for soundness, meaning that if a term is well-typed, it will always be well-typed. This is not only an important verification and safety guarantee for any programming language, but especially for shell languages, since runtime errors can become dangerous.

Example Consider a simple type system for a language with integer and boolean types. The type system includes rules for type checking expressions like arithmetic operations and comparisons. For example, an addition operation $a + b$ is well-typed if both a and b are integers, and the result is also an integer.

- **Progress**: If the expression $1 + 2$ is well-typed (as an integer), execution will proceed to the result 3 without getting stuck.
- **Preservation**: If $x + 1$ is well-typed and x steps to a value 5, then $5 + 1$ is also well-typed (as an integer), preserving the type correctness through execution.

By ensuring that a program that passes type checking adheres to its specified types during execution, soundness of the type system provides a robust foundation for building reliable, and efficient software.

2.4 Structural congruence

Structural congruence is a relation that defines when two syntactic expressions in a formal system can be considered equivalent in structure[8]. This relation is fundamental in the study of operational semantics and type systems, as it allows us to reason about programs by simplifying or rearranging expressions without changing their meaning. We denote structural congruence by $\equiv$.

Example An example of structural congruence in action is how the + operator works in mathematics. For the + operator, the following equivelences hold:

- **Commutativity:** The order of certain expressions can be swapped without affecting their equivalence.

$$e_1 \ + \ e_2 \equiv e_2 \ + \ e_1$$

.

- **Associativity:** Grouping of expressions can be changed without affecting their equivalence.

$$(e_1 \ + \ e_2) \ + \ e_3 \equiv e_1 \ + \ (e_2 \ + \ e_3)$$

- **Identity:** Certain expressions have an identity element that can be added or removed without affecting equivalence.

$$e \ + \ 0 \equiv e$$

where 0 is the identity element for the operator +.

Structural congruence ensures that programs can be reasoned about in a modular and flexible way, allowing transformations that preserve the semantics of the program. This is crucial for proving properties of programs. An important property as seen from a type system perspective is that structural equivalences must preserve the type of the expression in order to be valid in a strongly typed system. This is important since that is an obvious limitation of how equivalences can be used, and something we must remember when defining λ_{sh}.

2.5 Summary

In this chapter, we have laid the theoretical foundations essential for the development and understanding of the λ_{sh} language. Our exploration has covered several core areas, including formal operational semantics, type systems, and structural congruence, all of which contribute to the robustness and reliability of λ_{sh}.

We began by discussing formal operational semantics, which provide a rigorous mathematical framework for describing the behaviour of programs within λ_{sh}. By defining the syntax and the transition systems for our language, we ensured that the execution of commands and expressions is precise and unambiguous. This foundational work is critical for predicting and reasoning about the behaviour of λ_{sh} programs. The type system was another important part. We detailed how types are assigned to expressions within λ_{sh}, ensuring that operations are performed on compatible types. This prevents type errors and enhances the reliability of scripts. We introduced key concepts such as typing contexts, type judgements, and subtyping, which collectively enforce type safety and provide early error detection. The type system's soundness, guaranteed by the properties of preservation and progress, further ensures that well-typed programs do not encounter

type-related runtime errors. We also delved into structural congruence, a relation that allows us to reason about the equivalence of syntactic expressions. This concept is crucial for simplifying and rearranging expressions without altering their meaning, thereby supporting the modularity and flexibility of λ_{sh}.

In summary, the theoretical foundations presented in this chapter establish a comprehensive and rigorous basis for λ_{sh}. By integrating formal semantics and a robust type system, we have created a language that combines the flexibility and utility of traditional shell environments with the reliability and safety of modern type systems. These theoretical principles not only guide the design and implementation of λ_{sh} but also facilitate reasoning about its behaviour and properties, ensuring that it is dependable.

The theory presented here sets the stage for further exploration and development of λ_{sh}. As we move forward, these theoretical foundations will underpin our efforts to define the λ_{sh} language.

Chapter 3

Problem analysis

3.1 Introduction

The evolution of shell languages has significantly influenced the landscape of scripting and automation in computing environments[5]. Despite their widespread use and powerful capabilities, traditional shell languages like Bash and Zsh suffer from inherent limitations due to their weak type systems. These limitations often lead to runtime errors, security vulnerabilities, and maintenance challenges, making shell scripting a daunting task for developers. This chapter sets the stage for the development of λ_{sh}, a language that aims to integrate the formal foundations of λ-calculus with the practical utilities of traditional shell environments. The primary objective of λ_{sh} is to create a robust type system that enhances the reliability, security, and maintainability of shell scripts. By leveraging a strong type system, λ_{sh} seeks to mitigate the weaknesses of traditional shell scripting, providing developers with a more powerful and versatile scripting language.

We begin by discussing the core challenges associated with traditional shell languages, focusing on the implications of their weak type systems and the over-reliance on strings for data representation. We then introduce the concept of a strong type system and outline its numerous benefits, including type safety, early error detection, improved code clarity, optimisation opportunities, and enhanced security. The chapter explores the core concepts and definitions required for understanding the formal operational semantics of shell scripting. We explore how strings, numerals, and booleans can be modelled within a type system, and we define the syntactic categories and formation rules for these data types. Additionally, we present the semantic rules for string operations, demonstrating how a strong type system can handle implicit conversions and ensure type correctness.

By extending the semantics to support multiple data types and implementing coercion, we illustrate how λ_{sh} can provide a flexible scripting environment with sound foundations. Through detailed example derivations and soundness proofs, we validate the robustness of our proposed type system. This chapter provides the necessary foundation for developing λ_{sh} by addressing the challenges of weak type systems in traditional shell languages and demonstrating the benefits of strong typing. By laying this groundwork, we aim to pave the way for the creation of next-generation shell languages that combine the power and flexibility of traditional shells with the robustness and safety of modern type systems.

3.2 Challenges with traditional shell languages

In the domain of shell scripting, there are numerous elements that can be modelled formally through a type system and formal operational semantics. These elements include environment variables, file descriptors, the file system's state, process management, input/output redirections, command sequencing, conditional

execution, looping constructs, parallel execution, and more[2]. Each of these components plays a critical role in the behaviour and functionality of shell scripts, and formal modelling can provide a rigorous foundation for understanding and reasoning about their interactions.

Formal operational semantics allows us to define precise rules for how shell commands transition the state of the system. This includes how commands manipulate the current working directory, set and unset environment variables, handle input and output streams, launch and manage processes, and interact with the file system. By capturing these stateful mechanics, we can ensure that the behaviour of shell scripts is predictable and well-defined. Despite the wide range of aspects that can be modelled, we have chosen to focus our efforts on developing a strong type system for shell languages. We recognise the weak type systems typically found in shell scripting as a significant limitation. The lack of robust type-checking mechanisms often leads to runtime errors, security vulnerabilities, and maintenance challenges, which can hinder the reliability and efficiency of shell scripts. It is important to note that shells type systems can be considered so weak that they do not formally have a type system.

Traditional shell languages often allow for dynamic typing and implicit type conversions, which can result in subtle bugs that are difficult to detect and debug[6]. For instance, the incorrect handling of string and integer types, or the unintentional use of uninitialised variables, can lead to unexpected behaviours and script failures. These issues are compounded in large and complex scripts, where the lack of type safety can make code maintenance and refactoring particularly challenging. By introducing a strong type system, we aim to address these weaknesses and enhance the overall robustness of shell scripting. A strong type system can enforce type correctness at compile time, catching many errors early in the development process and reducing the likelihood of unexpected behaviours during execution. This not only improves the reliability of shell scripts but also facilitates easier debugging and maintenance by having type information available.

A strong type system can provide several benefits[11]:

- **Type safety**: Ensures that operations are performed on compatible types, preventing common errors such as type mismatches and invalid operations.
- **Early error detection**: Identifies type-related errors during the development phase, reducing the risk of encountering these errors at runtime.
- **Code clarity**: Enhances code readability and maintainability by making type constraints explicit, thereby reducing the cognitive load on developers.
- **Optimisation opportunities**: Enables compilers to generate more efficient machine code by leveraging type information, potentially improving script performance.
- **Security**: Mitigates certain types of vulnerabilities, such as injection attacks and buffer overflows, by enforcing strict type constraints. This property of course depends on the type system.

Moreover, a strong type system can support advanced language features such as type inference, polymorphism, and generic programming, further enhancing the expressiveness and flexibility of shell scripts. These features can make shell scripting more powerful and versatile, enabling developers to write more complex and robust scripts with greater ease. Although many of these features are a step beyond what we aim to do with this project, it is still important to have in mind what we can do once we have established the basis of a strong type system.

In summary, while there are many facets of the shell domain that can benefit from formal modelling, our decision to prioritise a strong type system is driven by the need to mitigate the inherent weaknesses of traditional shell scripting[2]. We believe that this approach will lead to more reliable, secure, and maintainable shell scripts, ultimately contributing to the broader adoption and effectiveness of shell languages

in various computing environments. By focusing on a strong type system, we aim to provide a solid foundation for the development of next-generation shell languages that combine the power and flexibility of traditional shells with the robustness and safety of modern type systems.

3.3 Weak Type Systems in Shell Languages

Shell languages, such as Bash, Zsh, and others, are widely used for scripting and automating tasks in Unix-like operating systems. Despite their power and flexibility, these languages often suffer from weak type systems, which can lead to various issues in script reliability, security, and maintainability[2].

A type system can be considered weak if it lacks strict type enforcement and fails to catch type-related errors at compile time[11]. Shell languages exhibit the following characteristics that contribute to their weak type systems[3]:

- **Dynamic typing**: Shell variables are dynamically typed, meaning their types are determined at runtime. This flexibility allows variables to change types based on the context of their usage. For example, a variable that holds an integer can later be used as a string without any explicit conversion, which can lead to unintended behaviour. This is of course because an integer is a string in typical shell languages.
- **Implicit type conversions**: Shell languages frequently perform implicit type conversions. For instance, arithmetic operations can automatically convert strings to integers, and vice versa, without explicit type declarations. This can result in subtle bugs, especially when dealing with user inputs or variable assignments.
- **Lack of type annotations**: Most shell languages do not support type annotations or declarations. This absence of explicit type information makes it difficult to reason about the types of variables and function arguments, leading to potential type mismatches and runtime errors.

The weak type systems in shell languages can lead to several significant issues. Since type errors are not caught at compile time, they often manifest as runtime errors, causing scripts to fail unpredictably and leading to downtime and other operational issues[2]. Additionally, weak typing can contribute to security vulnerabilities. A strong type system can enforce stricter type constraints, mitigating such risks. Scripts written with weak type systems can also be difficult to maintain and extend. Refactoring such scripts can introduce new bugs if type-related assumptions are violated. Moreover, debugging type-related issues in shell scripts can be challenging due to the dynamic nature of variable typing and implicit conversions. Consider the following examples to illustrate type-related issues in shell scripts:

```
# Example 1: Implicit type conversion
num="123"
sum=$((num + 10))
echo $sum  # Outputs 133

num="abc"
sum=$((num + 10))
echo $sum  # Causes a runtime error

# Example 2: Dynamic typing and type mismatches
var="Hello"
var=$((var + 1))
echo $var  # Causes a runtime error
```

In the first example, the variable `num` is implicitly converted to an integer for the arithmetic operation, which works as expected for numeric strings but causes a runtime error for non-numeric strings. In the second example, the variable `var` is treated as a string and later used in an arithmetic context, leading to a type mismatch and runtime error.

In summary, while shell languages provide powerful tools for automation and scripting, their weak type systems pose significant challenges for making safe scripts. By recognising and fixing these weaknesses, developers can improve the reliability, security, and maintainability of their shell scripts. Moving towards stronger type systems and better type-checking mechanisms will lead to more robust and error-free scripting environments, enhancing the effectiveness of shell languages in various computing contexts.

3.4 The special role of strings in shells

In addition to weak type systems, another significant challenge in traditional shell languages is the over-reliance on strings for data representation and manipulation. Strings are fundamental to shell programming, serving as the primary medium for specifying file paths, constructing commands, processing inputs, and representing various types of data[2]. While this string-centric approach simplifies language design and provides a uniform way to handle data, it also introduces complexities and limitations that can affect script functionality, readability, and maintainability.Therefore, understanding the nature of strings role in shell languages and string manipulation is essential for effective shell programming. This section explores the fundamental aspects of strings in shell programming, exploring their usage, manipulation techniques, and the challenges they pose.

Traditional shell languages, such as Bourne Again Shell, predominantly utilise strings as their fundamental data type[3]. The over reliance on strings can lead to a range of issues, particularly with string interpolation, quoting, and when strings are used to represent some distinct type of data. For instance, the treatment of white-space and special symbols often results in unpredictable behaviours if not meticulously managed, leading to crashes and hard-to-debug errors[6]. This is mainly due to the odd parsing strategies used by shells. The lack of strict data typing in traditional shells forces programmers to manually handle type conversions and validations, increasing the risk of runtime errors and security vulnerabilities. Additionally, this string-centric model hampers the ability to perform efficient static analysis, making it difficult to catch errors.

The over-reliance on strings also contributes to the opacity and complexity of shell scripts, as developers must constantly deal with the nuances of string manipulation and parsing[6]. This can result in scripts that are difficult to read, maintain, and extend, ultimately impacting the productivity and reliability of shell programming. To address these limitations, it is important to develop a better type system within shell languages. A robust type system would provide clearer semantics for different types of data, reducing the ambiguity inherent in treating everything as a string. By incorporating a more sophisticated type system, shell languages could enforce stricter type checks, catching errors at compile time rather than at runtime. This would not only improve the reliability and safety of shell scripts but also enhance their readability and maintainability.

3.5 Strings in Action

To demonstrate the crucial role of strings in shell programming, we will explore how commands work in typical shell languages, with a specific focus on string manipulation and how command interfaces are defined as strings.

3.5.1 Commands in Shell

In the context of Bash (Bourne Again SHell) and other shell languages, a "word" refers to a sequence of characters considered as a single unit by the shell[3]. Words are fundamental syntactic and semantic elements in shell programming. They can represent commands, arguments, or data inputs. Bash uses spaces (or other Internal Field Separator (IFS) characters) to separate words, treating each word as a distinct string. For example, consider the command:

```
cp myfile.txt backup/myfile.txt
```

In this command:

- `cp` is the first word, representing the copy command.
- `myfile.txt` is the second word, indicating the source file name.
- `backup/myfile.txt` is the third word, indicating the destination path and file name.

Each string separated by spaces is treated as a distinct word by the shell. Words can be grouped into a single word using quotes. This is particularly useful for handling filenames or arguments containing spaces. For instance:

```
cp "my file.txt" "backup/my file.txt"
```

Here, `"my file.txt"` and `"backup/my file.txt"` are treated as single words despite containing spaces, thanks to the enclosing quotes. This example highlights how strings, when quoted, can be preserved as single words, emphasizing the importance of understanding string and word boundaries in Bash. In Bash scripting, grasping how strings are parsed into words and how these words are processed is essential for accurately handling commands, arguments, and data manipulation. Commands are essentially compositions of strings. The syntax for defining a command is:

$$\langle \text{command} \rangle = \langle \text{string} \rangle +$$

In this context, we will not consider word expansion or variables, as these are extended features beyond fundamental command mechanics. A command is composed of one or more strings. A simple shell command like `mkdir dir1 dir2 dir3` is interpreted as a list of strings: `["mkdir", "dir1", "dir2", "dir3"]`, where the first string identifies the command. This identifier can be either a file path to an executable or the name of an executable within the shell's search path. Commands mirror the argument interface to processes in Unix, characterised by the `main` function's signature: `int main(int argc, char **argv) { ... }`. The entire list of strings constituting the command is passed to the executable.

The structure of commands includes:

1. The first string as the name or location of the executable command. This is either a direct file path or a command name recognised within the shell's environment path.
2. Subsequent strings are treated as arguments to the command, providing specifics on how the command operates.

This structure emphasises the modular and executable nature of commands within the shell, where commands are not only limited to named functions but also include executable files, all determined by their position and context within the command sequence. As established earlier, the first string in a command is the command name or identifier. The following order describes how the commands are looked up by the shell:

1. The shell looks up the command name in its list of built-in commands.
2. If not found, it searches for an executable in the directories listed in the `$PATH` environment variable or in the current working directory.
3. If not found, the shell reports an error.

When a command is defined with a command name, the shell runs the command with all the strings making up the command passed to the process that executes the command. The actual mechanics of running a command are beyond the scope of this section. It is important to note that the command name string is not a special string; it is just a string like any other.

3.6 Designing a string-centric type system

It should be obvious from our analysis of how commands work that strings play an important role in shell programming. They serve as the medium through which we communicate with external processes. Because of this, we propose designing our type system in such a way that strings are given special consideration. This string-centric approach is necessary to accurately reflect the nature of shell programming, where strings form the backbone of command interfaces and data exchange.

However, focusing on strings does not mean sacrificing the robustness of our type system. Our goal is to maintain a strong type system while accommodating the unique requirements of string handling in shell programming. Our observation is the following: in typical shells most pieces of data are strings until they are implicitly converted to something else doing an operation such as arithmetic. But why not do it in the other direction? Everything is the most particular datatype that it represents, and is converted to strings when needed. Most of the data worked with in shells, such as booleans and numbers, have neat string representations, so conversions are entirely possible. Then all data types that are representable as strings can be used as strings, as long as all operations that strictly require a string converts that piece of data to its string representation. Or maybe, we don't even need to convert anything to strings at all, if the data types can be used in string operations by only converting to strings when absolutely necessary. Operations such as string concatenation does not need the operation to actually convert anything to strings in order to work. Only when the string resulting value is used as a string, for example if it is being read.

In order to go about designing a type system with special focus on strings, we should explore how we can built such a type system and how it would materialise. We can start our exploration with what we want to do with strings. The most important string operations is string concatenation and string interpolation, and as we will explain in the next section, they are in fact the same.

3.7 String interpolation and concatenation

String interpolation is a programming technique that allows you to insert the value of variables or expressions into a string. It makes it easier to construct strings dynamically by embedding variable values directly within a string template. Different programming languages offer various ways to achieve string interpolation.

In Bash, string interpolation is straightforward and uses the $ symbol to insert the value of variables into strings[3]. An example can be seen in Listing 3.1.

```
# Define a variable
name="Nikolai"
age=24

# Simple string interpolation
echo "Hello, $name!"  # Output: Hello, Nikolai!

# Interpolating multiple variables
echo "$name is $age years old."  # Output: Nikolai is 24 years old.

# Using curly braces to avoid ambiguity
file_extension="txt"
echo "The file is named ${name}.${file_extension}"  # Output: The file is named
    Nikolai.txt

# Arithmetic operations within a string
number=5
echo "The double of $number is $((number * 2))."  # Output: The double of 5 is 10.
```

Listing 3.1: String Interpolation in Bash

Bash offers a straightforward method for string interpolation, but it is somewhat simplistic. This is because, in Bash, all values are fundamentally treated as strings. Bash lacks distinct data types, except in special cases where string values are implicitly converted, such as in arithmetic and boolean comparisons. There are also arrays and mappings, but these are rarely used.

Python provides a more sophisticated approach to string interpolation compared to Bash. Unlike Bash, Python supports multiple data types, yet its string interpolation syntax is similarly straightforward. This is achieved through underlying semantics that handle type conversions seamlessly[9]. In Python, there are several methods for string interpolation, including the % operator, the `str.format()` method, and f-strings (formatted string literals). These methods offer flexibility and power in constructing dynamic strings. We will be focusing on string interpolation of the f-string kind, since they are similar to how Bash string interpolation works.

Conventions, such as F-strings, provide a convenient and readable way to embed expressions inside string literals using curly braces in python and $ in Bash. However, you can achieve the same result using a string concatenation operator such as + in Python. This approach involves converting each expression to a string and concatenating it with other string parts. An example of this can bee seen in Listing 3.2 and Listing 3.3, which are fundamentally equivalent programs. In that sense, f-strings are essentially just syntactic sugar for string concatenation with automatic conversion to strings.

```
name = "Nikolai"
age = 24
greeting = f"Hello, {name}! You are {age} years old."
print(greeting)
```

Listing 3.2: String Interpolation using F-strings

```
name = "Nikolai"
age = 24
greeting = "Hello, " + str(name) + "! You are " + str(age) + " years old."
print(greeting)
```

Listing 3.3: String Interpolation using String Concatenation

In Listing 3.3, it is necessary to explicitly convert non-string values to strings using the `str()` function. This can make the code verbose and harder to read, especially when dealing with multiple variables and expressions. In contrast, f-strings, such as in Figure 3.2, provide a more elegant solution. They automatically handle the conversion of non-string values to strings, allowing for a more concise and readable syntax. We could imagine that we could make string concatenation work in a similar way by converting its arguments automatically into strings if such a representation is available for the datatype.

As we can see, string interpolation is just syntactic sugar for string concatenation with implicit conversions to strings. In essence, string interpolation allows for a more readable and convenient way to construct strings by embedding expressions directly within string literals. However, under the hood, this process is translated into a series of string concatenation operations. The operational semantics for this process involve evaluating each expression within the interpolation, converting the result to a string if necessary, and then performing the concatenation.

We will now explore a few semantic of how string concatenation with such behaviour can be defined formally in Section 3.8.

3.8 String concatenation and conversion semantics

In order to get a better insight into how string operations work, we explore a few simple string concatenation semantics inspired by those described in Section 3.7. We will start with the most basic one in Section 3.9 which resembles how we would view string concatenation in a language such as Bash. Then, we extend that semantics to support more data types in Section 3.10 and see how that changes the semantics. Finally, we use structural congruence to improve upon the semantics from Section 3.10 in Section 3.11.

3.8.1 Structural congruence

Before we introduce a concrete semantics for string operations, we want to look at some properties for the operations we want to perform. Specifically, we want to introduce some structural equivalences that are valid for string concatenation. Structural congruence, denoted as $\equiv$ defines when two expressions are considered equivalent in structure. For our string concatenation, we can define structural congruence rules for reflexivity, symmetry, transitivity, associativity of concatenation.

$$
\begin{aligned}
e \equiv e \quad & \text{(Reflexivity)} \\
\text{If } e_1 \equiv e_2, \text{ then } e_2 \equiv e_1 \quad & \text{(Symmetry)} \\
\text{If } e_1 \equiv e_2 \text{ and } e_2 \equiv e_3, \text{ then } e_1 \equiv e_3 \quad & \text{(Transitivity)} \\
(e_1 \circ e_2) \circ e_3 \equiv e_1 \circ (e_2 \circ e_3) \quad & \text{(Associativity)}
\end{aligned}
$$

Figure 3.1: Structural equivalences in for string concatenation.

It should be obvious that these properties hold for string concatenation operations. These properties are important when defining a programming language, since they have an impact on how one should define an evaluation strategy for a certain expression. Does it matter if we evaluate e_1 before e_2, or whether they switch places in a larger expression? That is some of the questions we get answered when defining the structural congruence's in a language. These properties hold for all of the languages we will present in this thesis.

3.9 String concatenation

The first semantics are simple rules without any other data types than strings. The general idea is to showcase how a string only languages would handle string operations, such that we can built upon this foundation later. In the language we have the following:

3.9.1 Syntactic categories and formation rules

$$x \in \mathbf{V}$$
$$s \in \mathbf{S}$$
$$e \in \mathbf{E}$$

Figure 3.2: Syntactic categories for the simple string concatenation language.

In Figure 3.12, **V** is the set of all variables, **S** is the set of all strings, and **E** is the set of all expressions.

$$e ::= x \mid s \mid e_1 \circ e_2$$

Figure 3.3: Formation rules for expressions.

As seen in Figure 3.3, the string concatenation operator is denoted as $\circ$.

3.9.2 Semantics

In order to define string concatenation we develop a big step semantics. To define a big step semantics of our language, we need to introduce the value category $v \in \mathbb{S}$, which encompasses all possible string values. This category allows us to formalise the behaviour and evaluation of strings within our system.

Our semantic model is represented as a transition system as a triple $(C, \to, T)$, where:

$$C = E \quad \text{(expressions)}$$
$$T = \mathbb{S} \quad \text{(type of values)}$$

In order to define our operational semantics we need to define two functions, σ and ϵ. σ is the representation of the environment in this simple calculi. σ maps from variables to strings $\sigma(x) \in \mathbb{S}$. ϵ is a function that maps elements from the syntactic category of strings **S** to the value category of strings $\mathbb{S}$. These functions are used in Figure 3.4.

$$\text{[VAR]} \qquad \frac{}{x \to v} \ \texttt{if}\ \sigma(x) = v$$

$$\text{[STRING]} \qquad \frac{}{s \to v} \ \texttt{if}\ \epsilon(s) = v$$

$$\text{[CONCAT]} \qquad \frac{e_1 \to v_1 \quad e_2 \to v_2}{e_1 \circ e_2 \to v} \ \texttt{if}\ v = v_1 \circ v_2$$

Figure 3.4: Big-step semantics for a simple string concatenation language.

The transition system is defined in Figure 3.4. The rule [VAR] states that a variable x transitions to a value v if the variable x is mapped to the value v in the environment σ. Essentially, this rule retrieves the value associated with the variable from the environment. The rule [STRING] says that a string literal s transitions to a value v if the evaluation of s using the mapping function ϵ results in v. This rule directly maps string literals to their corresponding values. The rule [CONCAT] handles the concatenation of two expressions e_1 and e_2 transitions to the value v if e_1 transitions to the value v_1 and e_2 transitions to the value e_2, and the concatenation of v_1 and v_2 results in v.

3.9.3 Type rules

To define the type system for our language, we introduce a set of type rules that ensure the syntactic expressions are correctly typed. These rules enforce that operations on strings are valid and that the resulting types are consistent within the semantic framework. Now, these are very simple right now since we only have strings, but we will expand on them shortly. The type rules for simple string operations are summarised in Figure 3.5.

The typing context Γ is a mapping from variables to their types. It provides the necessary information to type-check expressions involving variables. For example, if $\Gamma(x) = \mathit{String}$ it means that the variable x is known to have the type String within this context. In this simple language we have that $\forall x \in \mathbf{V}.\Gamma(x) = \mathit{String}$ since Γ is a total mapping from variables to strings.

$$\text{[T-VAR]} \quad \frac{\Gamma(x) = \mathit{String}}{\Gamma \vdash x : \mathit{String}}$$

$$\text{[T-STRING]} \quad \frac{}{\Gamma \vdash s : \mathit{String}}$$

$$\text{[T-CONCAT]} \quad \frac{\Gamma \vdash e_1 : \mathit{String} \quad \Gamma \vdash e_2 : \mathit{String}}{\Gamma \vdash e_1 \circ e_2 : \mathit{String}}$$

Figure 3.5: Type rules for string concatenation.

In Figure 3.5, we have the type rules. The rule [T-VAR] states that a variable x is of type *String* if x is present in the typing context Γ with the type *String*. The rule [T-STRING] states that a string literal s is always of type *String*, regardless of the context. The rule [T-CONCAT] states that the concatenation of two expressions e_1 and e_2 is of type String if both e_1 and e_2 are of type *String* in the typing context Γ. We will base our later type systems on the same system of a typing context Γ. This description outlines the complete semantics for a basic form of string concatenation, assuming all values involved are strings. An example of a language that utilises this type of string concatenation is Bash, as discussed in Section 3.7. Now, let us expand on this to include more than just strings as data types.

As we can see with this small-semantics, it is not particularly difficult to built a semantics for string operations when we are only dealing with strings as in languages like Bash, but what if we are not just dealing with strings like in Python, where we have multiple data types? Then we still want to deal with string operations as easily as in this small semantics, but we obviously need other systems or mechanics to handle this.

3.10 String concatenation extended with more data types

We will now expand the language with numerals and booleans, which are types that are easily representable as strings. To expand the syntax to include numerals and booleans, we need to extend the syntactic categories and the formation rules to incorporate these additional types. Additionally, we'll modify the semantics to handle implicit conversions of numbers and booleans to strings during concatenation.

We will introduce new syntactic categories for numerals and booleans, and define new formation rules. We will also update the semantics to handle these new types and their implicit conversions.

3.10.1 Syntactic categories and formation rules

We extend the syntactic categories from Figure 3.12 with the categories from Figure 3.6.

$$x \in \mathbf{V}$$
$$s \in \mathbf{S}$$
$$n \in \mathbf{N}$$
$$b \in \mathbf{B}$$
$$e \in \mathbf{E}$$

Figure 3.6: Caption

In Figure 3.6, $\mathbf{N}$ is the set of all numerals and $\mathbf{B}$ is the set of all booleans. Furthermore, we extend the formation rules in 3.3 as described Figure 3.7.

$$e ::= x \mid s \mid n \mid b \mid e_1 \circ e_2$$

Figure 3.7: Formation rules for expressions.

3.10.2 Semantics

To define the semantics of the expanded language, we need to handle the implicit conversion of numerals and booleans to strings during concatenation. We update our semantic model and introduce the necessary functions and rules. First, the value category $v \in \mathbb{V}$ now includes strings, numerals, and booleans, where $\mathbb{N}$ is the set of numerals values and $\mathbb{B}$ is the set of boolean values:

$$\mathbb{s} \in \mathbb{S}$$
$$\mathbb{n} \in \mathbb{N}$$
$$\mathbb{b} \in \mathbb{B}$$
$$\mathbb{v} \in \mathbb{V} = \mathbb{S} \cup \mathbb{N} \cup \mathbb{B}$$

Our semantics model remains a transition system represented as a triple $(C, \to, T)$, where:

$$C = \mathbf{E} \quad \text{(expressions)}$$
$$T = \mathbb{V} \quad \text{(type of values)}$$

We redefine our functions σ and ϵ to handle the new types. Specifically, Γ maps variables to values in $\mathbb{V}$, and ϵ maps elements from the syntactic categories **S**, **N**, and **B** to the equivalent value in the value category $\mathbb{V}$.

$$\text{[VAR]} \quad \frac{}{x \to v} \texttt{ if } \sigma(x) = v$$

$$\text{[STRING]} \quad \frac{}{s \to \mathbb{s}} \texttt{ if } \epsilon(s) = \mathbb{s}$$

$$\text{[NUM]} \quad \frac{}{n \to \mathbb{n}} \texttt{ if } \epsilon(n) = \mathbb{n}$$

$$\text{[BOOL]} \quad \frac{}{b \to \mathbb{b}} \texttt{ if } \epsilon(b) = \mathbb{b}$$

$$\text{[CONCAT]} \quad \frac{e_1 \to v_1 \quad e_2 \to v_2}{e_1 \circ e_2 \to \mathbb{s}} \texttt{ if } \mathbb{s} = \texttt{toString}(v_1) \circ \texttt{toString}(v_2)$$

Figure 3.8: Semantic rules for the expanded string concatenation language.

The transition system is defined in Figure 3.8. The rule [VAR] states that a variable x transitions to a value v if the variable x is mapped to the value v in the environment Γ. The rule [STRING] states that a string literal s transitions to a value v if the evaluation of s using the mapping function ϵ results in v. The rule [NUM] states that a numeral n transitions to a value v if the evaluation of n using the mapping function ϵ results in v. The rule [BOOL] states that a boolean b transitions to a value v if the evaluation of b using the mapping function ϵ results in v. The rule [CONCAT] handles the concatenation of two expressions e_1 and e_2, and it ensures that the values v_1 and v_2 are implicitly converted to strings before concatenation. The `toString` function is a mapping from values $v \in \mathbb{V}$ to their string representations in the value category $\mathbb{S}$. For values in $\mathbb{S}$, this function serves as an identity function. How exactly this conversion from values in $\mathbb{N}$ and $\mathbb{B}$ is made is considered an implementation detail.

3.10.3 Type system

To define the type system for our expanded language, we introduce new type rules that ensure the syntactic expressions involving numerals and booleans are correctly typed. These rules enforce that operations on strings, numerals, and booleans are valid and that the resulting types are consistent within the semantics of the type system. We update the typing context Γ such that we can type-check expressions involving variables. The typing context Γ is defined as a total mapping from variables $\mathbf{V}$ to types $\{\texttt{Bool}, \texttt{String}, \texttt{Num}\}$. This means that for any variable $x \in \mathbf{V}$, $\Gamma(x)$ will specify whether x is of type `Bool`, `String`, or `Num`. For example, if $\Gamma(x) = \texttt{String}$, it means that the variable x is known to have the type `String` within this context.

$$\text{[T-VAR]} \quad \frac{\Gamma(x) = T \quad T \in \{\texttt{String}, \texttt{Num}, \texttt{Bool}\}}{\Gamma \vdash x : T}$$

$$\text{[T-STRING]} \quad \frac{}{\Gamma \vdash s : \texttt{String}}$$

$$\text{[T-NUM]} \quad \frac{}{\Gamma \vdash n : \texttt{Num}}$$

$$\text{[T-BOOL]} \quad \frac{}{\Gamma \vdash b : \texttt{Bool}}$$

$$\text{[T-CONCAT]} \quad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1, T_2 \in \{\texttt{String}, \texttt{Num}, \texttt{Bool}\}}{\Gamma \vdash e_1 \circ e_2 : \texttt{String}}$$

Figure 3.9: Type rules for the expanded string concatenation language.

In Figure 3.9, we have the updated type rules. The rule [T-VAR] states that a variable x is of type T if x is present in the typing context Γ with the type T, where T can be `String`, `Num`, or `Bool`. The rule [T-STRING] states that a string literal s is always of type `String`. The rule [T-NUM] states that a numeral n is always of type `Num`. The rule [T-BOOL] states that a boolean b is always of type `Bool`. The rule [T-CONCAT] states that the concatenation of two expressions e_1 and e_2 is of type `String` if both e_1 and e_2 are of types that can be converted to strings (`String`, `Num`, or `Bool`) in the typing context Γ.

3.10.4 Example derivation

In this example we create a derivation of the type system and semantics in order to provide a better intuition of how the rules are applied. We use the example *"You are "* $\circ$ 24 $\circ$ *" years old!"*.

First, it is shown in the derivation using the type rules in Figure 3.10, which demonstrates that *"You are "* $\circ$ 24 $\circ$ *" years old!"* is well-typed.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \textit{"You are "} : \texttt{String}}\,\texttt{[T-STRING]} \quad \dfrac{}{\Gamma \vdash 24 : \texttt{Num}}\,\texttt{[T-NUM]}}{\Gamma \vdash (\textit{"You are "} \circ 24) : \texttt{String}}\,\texttt{[T-STRING]} \quad \dfrac{}{\Gamma \vdash \textit{" years old!"} : \texttt{String}}\,\texttt{[T-STRING]}}{\Gamma \vdash (\textit{"You are "} \circ 24 \circ \textit{" years old!"}) : \texttt{String}}\,\texttt{[T-CONCAT]}$$

Figure 3.10: Type derivation for string concatenation with multiple data types.

Secondly, we show in Figure 3.11 that there exists a derivation for *"You are "* $\circ$ 24 $\circ$ *" years old!"* such that the program evaluate down to *" You are* 24 *years old!*".

$$\dfrac{\dfrac{\dfrac{}{\textit{"You are "} \to \textit{"You are "}}\,\texttt{[STRING]} \quad \dfrac{}{24 \to 24}\,\texttt{[NUM]}}{\textit{"You are "} \circ 24 \to \textit{"You are 24"}}\,\texttt{[CONCAT]} \quad \dfrac{}{\textit{" years old!"} \to \textit{" years old!"}}\,\texttt{[STRING]}}{\textit{"You are "} \circ 24 \circ \textit{" years old!"} \to \textit{"You are 24 years old!"}}\,\texttt{[CONCAT]}$$

Figure 3.11: Example derivation for string concatenation with multiple data types.

As we have seen Figure 3.10 and Figure 3.11, the type system seems well-defined and the semantics

seem to work too. Now, we want to show that the type system is sound, such that we can confidently say the type system and semantics are well-defined.

3.10.5 Soundness proof

A type system is considered sound if it guarantees that programs which have been successfully type-checked will not produce certain kinds of runtime errors related to types. In other words, if a program is well-typed according to the rules of the type system, it will not encounter type errors when it is executed. This concept is crucial in programming languages because it provides a form of correctness assurance, meaning that type-related errors can be caught at compile time rather than at runtime. To prove the soundness of the type system for the expanded string concatenation language with numerals and booleans, we need to show that well-typed programs do not go wrong. This involves demonstrating two key properties:

- Preservation: If a well-typed expression takes a step in the operational semantics, the resulting expression is also well-typed.
- Progress: A well-typed expression is either a value or can take a step according to the operational semantics.

We start by showing the preservation property.

Theorem 1 (Preservation: the reduction rules preserves types)**.** *Assume that e is an expression, if* $\Gamma \vdash e : \tau$ *and* $e \to e'$*, then* $\Gamma \vdash e' : \tau$*.*

Proof. We proceed by induction on the derivation of the evaluation $e \to e'$.

Case [VAR]:

$$\frac{}{x \to v} \texttt{ if } \sigma(x) = v$$

From the typing rule `[T-VAR]`, we have $\Gamma \vdash x : T$ where $\Gamma(x) = T$. Since $x \to v$ by looking up x in σ, and assuming the environment σ maps variables to values of the correct type, we have $\Gamma \vdash v : T$.

Case [STRING]:

$$\frac{}{s \to \mathbb{s}} \texttt{ if } \epsilon(s) = \mathbb{s}$$

From the typing rule `[T-STRING]`, we have $\Gamma \vdash s$: `String`. Since $s \to \mathbb{s}$ and $\mathbb{s}$ is the string value corresponding to s, $\Gamma \vdash \mathbb{s}$: `String` by definition.

Case [NUM]:

$$\frac{}{n \to \mathbb{n}} \texttt{ if } \epsilon(n) = \mathbb{n}$$

From the typing rule `[T-NUM]`, we have $\Gamma \vdash n$: `Num`. Since $n \to \mathbb{n}$ and $\mathbb{n}$ is the numeral value corresponding to n, $\Gamma \vdash \mathbb{n}$: `Num` by definition.

Case [BOOL]:

$$\frac{}{b \to \mathbb{b}} \texttt{ if } \epsilon(b) = \mathbb{b}$$

From the typing rule `[T-BOOL]`, we have $\Gamma \vdash b$: `Bool`. Since $b \to \mathbb{b}$ and $\mathbb{b}$ is the boolean value corresponding to b, $\Gamma \vdash \mathbb{b}$: `Bool` by definition.

Case [CONCAT]:

$$\frac{e_1 \to v_1 \quad e_2 \to v_2 \quad \mathtt{s} = \mathtt{toString}(v_1) \circ \mathtt{toString}(v_2)}{e_1 \circ e_2 \to \mathtt{s}}$$

From the typing rule [T-CONCAT], we have $\Gamma \vdash e_1 \circ e_2 : \mathtt{String}$ if $\Gamma \vdash e_1 : T_1$ and $\Gamma \vdash e_2 : T_2$ where $T_1, T_2 \in \{\mathtt{String}, \mathtt{Num}, \mathtt{Bool}\}$.

Assume $e_1 \to v_1$ and $e_2 \to v_2$. By the induction hypothesis, $\Gamma \vdash v_1 : T_1$ and $\Gamma \vdash v_2 : T_2$. Since v_1 and v_2 are values, $\mathtt{toString}(v_1)$ and $\mathtt{toString}(v_2)$ produce string values. Thus, $\mathtt{s} = \mathtt{toString}(v_1) \circ \mathtt{toString}(v_2)$ is a string, and hence $\Gamma \vdash \mathtt{s} : \mathtt{String}$.

□

Now that we have shown the preservation property, we show the progress property.

Theorem 2 (Progress: an expression is either a value or can be evaluated further)**.** *Assume that e is and expression. If $\Gamma \vdash e : \tau$, then e is either a value or there exists an e' such that $e \to e'$.*

Proof. We proceed by induction on the derivation of the typing judgment $\Gamma \vdash e : T$.

Case [T-VAR]:

$$\frac{\Gamma(x) = T \quad T \in \{\mathtt{String}, \mathtt{Num}, \mathtt{Bool}\}}{\Gamma \vdash x : T}$$

Variables are assumed to be mapped to values in the environment σ, so they can take a step to their corresponding value.

Case [T-STRING]:

$$\frac{}{\Gamma \vdash s : \mathtt{String}}$$

A string literal is already a value.

Case [T-NUM]:

$$\frac{}{\Gamma \vdash n : \mathtt{Num}}$$

A numeral literal is already a value.

Case [T-BOOL]:

$$\frac{}{\Gamma \vdash b : \mathtt{Bool}}$$

A boolean literal is already a value.

Case [T-CONCAT]:

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \quad T_1, T_2 \in \{\mathtt{String}, \mathtt{Num}, \mathtt{Bool}\}}{\Gamma \vdash e_1 \circ e_2 : \mathtt{String}}$$

If e_1 is not a value, then by the induction hypothesis, there exists e_1' such that $e_1 \to e_1'$. Hence, $e_1 \circ e_2 \to e_1' \circ e_2$. Similarly, if e_2 is not a value, then $e_2 \to e_2'$ and $e_1 \circ e_2 \to e_1 \circ e_2'$. If both e_1 and e_2 are values, then $e_1 \circ e_2 \to v$ where v is a string.

□

Conclusion Since both preservation and progress properties hold, we have shown that the type system for the expanded string concatenation language with numerals and booleans is sound. Well-typed expressions do not go wrong: they either evaluate to a value of the correct type or can take a step according to the operational semantics.

3.11 String concatenation with subtyping and coercion

While the semantics defined in Section 3.10 is satisfactory, we can improve upon the generality of the conversion to strings by introducing a coercion rule and the use of subtyping.

3.11.1 Syntactic Categories and Formation Rules

To define string concatenation through structural congruence, we use the same syntactic categories as those in Figure 3.6.

$$x \in \mathbf{V}$$
$$s \in \mathbf{S}$$
$$n \in \mathbf{N}$$
$$b \in \mathbf{B}$$
$$e \in \mathbf{E}$$

Figure 3.12: Formation rules for the language.

$$e ::= x \mid s \mid n \mid b \mid e_1 \circ e_2$$

Figure 3.13: Extended formation rules for structural congruence.

3.11.2 Semantics

In the semantics of our language with multiple data types , we use most of the same definitions for σ, ϵ, and `toString` from Section 3.10. Our values also remain the same as in Section 3.10. Our transition system $(C, \to, T)$ is also the same, except that we introduce new transition rules $\to$ as seen in Figure 3.14.

$$\text{[VAR]} \quad \frac{}{x \to v} \ \texttt{if}\ \sigma(x) = v$$

$$\text{[STRING]} \quad \frac{}{s \to \mathbb{s}} \ \texttt{if}\ \epsilon(s) = \mathbb{s}$$

$$\text{[NUM]} \quad \frac{}{n \to \mathbb{n}} \ \texttt{if}\ \epsilon(n) = \mathbb{n}$$

$$\text{[BOOL]} \quad \frac{}{b \to \mathbb{b}} \ \texttt{if}\ \epsilon(b) = \mathbb{b}$$

$$\text{[CONCAT]} \quad \frac{e_1 \to \mathbb{s}_1 \quad e_2 \to \mathbb{s}_2}{e_1 \circ e_2 \to \mathbb{s}} \ \texttt{if}\ \mathbb{s} = \mathbb{s}_1 \circ \mathbb{s}_2$$

$$\text{[TO-STRING]} \quad \frac{}{\mathbb{v} \to \mathbb{s}} \ \text{if}\ \mathbb{s} = \textsc{toString}(\mathbb{v})$$

Figure 3.14: Semantic rules for the expanded string concatenation language with structural congruence.

The semantics presented in Figure 3.14 define how expressions in an expanded string concatenation language with structural congruence are evaluated. According to the [VAR] rule, variables are evaluated based on their assignments in the environment; if a variable x is assigned the value v in the environment σ, then x evaluates to v. For literals, the evaluation is straightforward: according to the [STRING] rule, a string s maps to its corresponding value $\mathbb{s}$, the [NUM] rule specifies that a numeric literal n maps to $\mathbb{n}$, and the [BOOL] rule states that a boolean literal b maps to $\mathbb{b}$, all based on the interpretation function ϵ. String concatenation is handled by the [CONCAT] rule, which evaluates the sub-expressions individually; if e_1 evaluates to $\mathbb{s}_1$ and e_2 evaluates to $\mathbb{s}_2$, then the concatenation of e_1 and e_2 results in the string $\mathbb{s}_1 \circ \mathbb{s}_2$. Coercion between types is managed by the [TO-STRING] rule, which indicates that if $\mathbb{s}$ is the string representation of the value $\mathbb{v}$, then $\mathbb{v}$ evaluates to the string $\mathbb{s}$.

These rules collectively provide a consistent framework for evaluating expressions in this language, ensuring that variables, literals, concatenations, and type coercion are interpreted correctly.

3.11.3 Type System

Formally, the set of types τ can be defined as:

$$\tau ::= \texttt{String} \mid \texttt{Num} \mid \texttt{Bool}$$

These types ensure that expressions within the language are type-safe and that type conversions are well-defined and consistent.

We introduce subtyping in order to allow values that are not strings but are representable as strings through a coercion to be used in places where strings are required. The subtyping relation $A <: B$ indicates that type A is a subtype of type B. For our language, we define the subtyping relation as follows:

$$\begin{aligned} &\texttt{Num} <: \texttt{String} \\ &\texttt{Bool} <: \texttt{String} \\ &\tau <: \tau \quad \text{(reflexivity)} \end{aligned}$$

This means:

1. Any numeric type (`Num`) can be considered a subtype of `String`.
2. Any boolean type (`Bool`) can be considered a subtype of `String`.
3. Any type is a subtype of itself (reflexivity).

$$\text{[T-VAR]} \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\text{[T-STRING]} \quad \frac{}{\Gamma \vdash s : \texttt{String}}$$

$$\text{[T-NUM]} \quad \frac{}{\Gamma \vdash n : \texttt{Num}}$$

$$\text{[T-BOOL]} \quad \frac{}{\Gamma \vdash b : \texttt{Bool}}$$

$$\text{[T-CONCAT]} \quad \frac{\Gamma \vdash e_1 : \texttt{String} \quad \Gamma \vdash e_2 : \texttt{String}}{\Gamma \vdash e_1 \circ e_2 : \texttt{String}}$$

$$\text{[T-SUB]} \quad \frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

Figure 3.15: Type rules for the expanded string concatenation language with structural congruence.

The type rules for the expanded string concatenation language through coercion provide a framework for ensuring the type correctness of various expressions within the language. According to the [T-VAR] rule, if the type environment Γ assigns a type τ to a variable x, then x has type τ. This ensures that variables are used consistently with the types assigned to them in the type environment. The [T-STRING] rule asserts that a string literal s is of type `String`. This rule guarantees that all string literals are recognised as having the correct string type within the type system. Similarly, the [T-NUM] rule specifies that a numeric literal n is of type `Num`. This rule ensures that numeric literals are correctly typed as numbers. The [T-BOOL] rule states that a boolean literal b is of type Boolean. This rule confirms that boolean literals are appropriately typed as booleans within the type system. The [T-CONCAT] rule handles the type of string concatenation expressions. It states that if both expressions e_1 and e_2 are of type `String`, then their concatenation $e_1 \circ e_2$ is also of type `String`. This ensures that the result of concatenating two strings is itself a string. The [T-SUB] rule introduces subtyping into the type system, allowing for more flexible type checking. It states that if an expression e has type τ_1, and τ_1 is a subtype of τ_2 (denoted $\tau_1 <: \tau_2$), then e can be assigned the type τ_2.

These rules collectively provide a consistent type system for the expanded string concatenation language, ensuring that variables, literals, concatenations, and type coercion are correctly typed and managed within the language framework.

3.11.4 Example Derivation

To illustrate how the subtyping relation and type rules work, as in Section 3.10, we create a derivation of the type system and semantics to provide a better intuition of how the rules are applied. We use the example $24 \circ$ *" years old!"* (The example from Section 3.10 is too long).

First, we show the type derivation using the type rules in Figure 3.16:

$$\dfrac{\dfrac{\Gamma \vdash 24 : \texttt{Num} \quad \texttt{Num} <: \texttt{String}}{\Gamma \vdash 24 : \texttt{String}}\ \texttt{[T-SUB]} \quad \dfrac{}{\Gamma \vdash \textit{" years old!"} : \texttt{String}}\ \texttt{[T-STRING]}}{\Gamma \vdash 24 \circ \textit{" years old!"} : \texttt{String}}\ \texttt{[T-CONCAT]}$$

Figure 3.16: Type derivation for $24 \circ$ *" years old!"*.

Next, we show the semantic derivation in Figure 3.17, illustrating that the expression evaluates to "24 *years old!*".

$$\dfrac{\dfrac{\dfrac{24 \to 24}{\texttt{toString}(24) \to \text{"24"}}\ \texttt{[TO-STRING]} \quad 24 \equiv \texttt{toString}(24)}{24 \to \text{"24"}}\ \texttt{[STRUCT]} \quad \dfrac{}{\text{" years old!"} \to \text{" years old!"}}\ \texttt{[STRING]}}{24 \circ \text{" years old!"} \to \text{"24 years old!"}}\ \texttt{[CONCAT]}$$

Figure 3.17: Semantic derivation for $24 \circ$ *" years old!"*.

As in Section 3.10, we now show a soundness proof for the type system. We want to prove that this type system is actually sound, since structural congruence introduces a more generalised version of implicit type conversion.

3.11.5 Soundness proof

To prove the soundness of the type system for the expanded string concatenation language with structural congruence, we need to show that well-typed programs do not go wrong. This involves demonstrating the two key properties:

- **Preservation**: If a well-typed expression takes a step in the operational semantics, the resulting expression is also well-typed.
- **Progress**: A well-typed expression is either a value or can take a step according to the operational semantics.

We start by showing the preservation property.

Theorem 3 (Preservation: the reduction rules preserves types)**.** *Assume e is and expression. If $\Gamma \vdash e : \tau$ and $e \to e'$, then $\Gamma \vdash e' : \tau$.*

Proof. We proceed by induction on the derivation of the operational semantics $e \to e'$.

Case [VAR]:

$$\dfrac{}{x \to v}\ \texttt{if}\ \sigma(x) = v$$

Given $\Gamma \vdash x : \tau$ and $x \to v$, we need to show $\Gamma \vdash v : \tau$. By the `[T-VAR]` rule, $\Gamma(x) = \tau$, and since x evaluates to v, v must have type τ.

Case [STRING]:

$$\dfrac{}{s \to \mathbb{s}}\ \texttt{if}\ \epsilon(s) = \mathbb{s}$$

Given $\Gamma \vdash s : \texttt{String}$ and $s \to \mathbb{s}$, we need to show $\Gamma \vdash \mathbb{s} : \texttt{String}$. By the `[T-STRING]` rule, s is a string literal and $\mathbb{s}$ is its value, which is of type `String`.

Case [NUM]:

$$\dfrac{}{n \to \mathbb{n}}\ \texttt{if}\ \epsilon(n) = \mathbb{n}$$

Given $\Gamma \vdash n : \texttt{Num}$ and $n \to \mathbb{n}$, we need to show $\Gamma \vdash \mathbb{n} : \texttt{Num}$. By the `[T-NUM]` rule, n is a numeric literal and $\mathbb{n}$ is its value, which is of type `Num`.

Case [BOOL]:

$$\frac{}{b \to \mathbb{b}} \ \texttt{if}\ \epsilon(b) = \mathbb{b}$$

Given $\Gamma \vdash b : \texttt{Bool}$ and $b \to \mathbb{b}$, we need to show $\Gamma \vdash \mathbb{b} : \texttt{Bool}$. By the `[T-BOOL]` rule, b is a boolean literal and $\mathbb{b}$ is its value, which is of type `Bool`.

Case [CONCAT]:

$$\frac{e_1 \to \mathbb{s}_1 \quad e_2 \to \mathbb{s}_2}{e_1 \circ e_2 \to \mathbb{s}} \ \texttt{if}\ \mathbb{s} = \mathbb{s}_1 \circ \mathbb{s}_2$$

Given $\Gamma \vdash e_1 \circ e_2 : \texttt{String}$ and $e_1 \circ e_2 \to \mathbb{s}$, we need to show $\Gamma \vdash \mathbb{s} : \texttt{String}$. By the premises, $e_1 \to \mathbb{s}_1$ and $e_2 \to \mathbb{s}_2$, and by induction hypothesis, $\Gamma \vdash \mathbb{s}_1 : \texttt{String}$ and $\Gamma \vdash \mathbb{s}_2 : \texttt{String}$. Therefore, $\mathbb{s} = \mathbb{s}_1 \circ \mathbb{s}_2$ is a string, and $\Gamma \vdash \mathbb{s} : \texttt{String}$.

Case [TO-STRING]:

$$\frac{}{\mathbb{v} \to \mathbb{s}} \ \text{if}\ \mathbb{s} = \texttt{toString}(\mathbb{v})$$

Given $\Gamma \vdash \mathbb{v} : \tau$ and $\mathbb{v} \to \mathbb{s}$, we need to show $\Gamma \vdash \mathbb{s} : \texttt{String}$. By the subtyping relation, $\tau <: \texttt{String}$, and thus, $\mathbb{s}$ is the string representation of $\mathbb{v}$, which means $\Gamma \vdash \mathbb{s} : \texttt{String}$.

□

Now that we have proved the preservation property, we need to show the progress property.

Theorem 4 (Progress: an expression is either a value or can be evaluated further)**.** *Assume that e is and expression. If $\Gamma \vdash e : \tau$, then e is either a value or there exists an e' such that $e \to e'$.*

Proof. We proceed by induction on the derivation of the typing judgement $\Gamma \vdash e : T$.

Case [T-VAR]:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Variables are assumed to be mapped to values in the environment σ, so they can take a step to their corresponding value.

Case [T-STRING]:

$$\frac{}{\Gamma \vdash s : \texttt{String}}$$

A string literal is already a value.

Case [T-NUM]:

$$\frac{}{\Gamma \vdash n : \texttt{Num}}$$

A numeral literal is already a value.

Case [T-BOOL]:

$$\frac{}{\Gamma \vdash b : \texttt{Bool}}$$

A boolean literal is already a value.

Case [T-CONCAT]:

$$\frac{\Gamma \vdash e_1 : \texttt{String} \quad \Gamma \vdash e_2 : \texttt{String}}{\Gamma \vdash e_1 \circ e_2 : \texttt{String}}$$

If e_1 is not a value, then by the induction hypothesis, there exists e_1' such that $e_1 \to e_1'$. Hence, $e_1 \circ e_2 \to e_1' \circ e_2$. Similarly, if e_2 is not a value, then $e_2 \to e_2'$ and $e_1 \circ e_2 \to e_1 \circ e_2'$. If both e_1 and e_2 are values, then $e_1 \circ e_2 \to v$ where v is a string.

Case [T-SUB]:

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

By the induction hypothesis, if $\Gamma \vdash e : \tau_1$, then e is either a value or there exists e' such that $e \to e'$. Since $\tau_1 <: \tau_2$, we consider the possible subtyping relations:

- If $\tau_1 = \texttt{Num}$ and $\tau_2 = \texttt{String}$, then by the subtyping relation $\texttt{Num} <: \texttt{String}$, we know that e can take a step according to the `[TO-STRING]` rule.
- If $\tau_1 = \texttt{Bool}$ and $\tau_2 = \texttt{String}$, then by the subtyping relation $\texttt{Bool} <: \texttt{String}$, we know that e can take a step according to the `[TO-STRING]` rule.

Therefore, in both cases, e can take a step to $\texttt{toString}(e)$ which is of type `String`. Thus, the progress property holds for τ_2.

□

Conclusion Since both preservation and progress properties hold, we have shown that the type system for the expanded string concatenation language with structural congruence is sound. Well-typed expressions do not go wrong: they either evaluate to a value of the correct type or can take a step according to the operational semantics.

3.12 Next

Now that we have showcased how a type system can be built around a string-centric model, we will describe how we have implemented λ_{sh} with this in mind. This implementation ensures that the type system uses subtyping to allow data types that can be represented as strings to be treated as strings. We have focused on creating a robust framework that maintains the simplicity and familiarity of traditional shell syntax while providing the benefits of a strong type system.

Chapter 4

Formal specification of λ_{sh}

In this section, we introduce λ_{sh}, a language designed to illustrate how we can model a shell language with a sound type system. λ_{sh} extends the traditional λ-calculus by integrating features commonly found in Unix-like shell environments, such as string interpolation, incorporating core operating system functions, representing the file system, and a highly concurrent environment.

The main goal of λ_{sh} is to demonstrate how these features can be formalised within a theoretical framework, providing a foundation for implementing a shell programming language. By combining the rigour of λ-calculus with practical elements of shell programming, λ_{sh} is supposed to serve as a bridge between theoretical computer science and real-world shell programming. We will begin by defining the syntactic categories and formation rules of λ_{sh}, followed by an exploration feature by feature of its operational semantics, type rules, and structural equivalences, concluding with a discussion on the properties of the type system and formal proofs of soundness.

4.1 Computational Model

The λ_{sh} language is based on the $\lambda\sigma$-calculus presented in the paper "Explicit Substitutions" by Abadi, Cardelli, Curien, and Lévy[1]. The $\lambda\sigma$-calculus refines the classical λ-calculus by making substitutions explicit, providing a framework for studying substitution theory with robust mathematical properties. This explicit handling of substitutions allows for a detailed understanding and verification of the implementation of λ-calculus, and serves as a good foundation of an implementation of a language based on λ-calculus, since it is well-suited for usage within a small-step semantics. The λ_{sh} language leverages these properties to manage and manipulate substitutions directly, ensuring precise and efficient computation.

4.2 Syntactic Categories and formation Rules

The λ_{sh} language extends the traditional λ-calculus by introducing features such as quoting expressions, core operating system functions, and a representation of the file system. These extensions allow λ_{sh} to effectively handle shell scripting tasks while maintaining the formal rigour of λ-calculus. This section defines the syntactic categories and formation rules of λ_{sh}, which form the foundation of the language's structure and semantics. The language is defined by the syntactic categories in Figure 4.1.

$$
\begin{aligned}
n &\in \mathbf{N} && \text{(Numerals)} \\
b &\in \mathbf{B} && \text{(Booleans)} \\
s &\in \mathbf{S} && \text{(String)} \\
p &\in \mathbf{P} && \text{(Filepath)} \\
o &\in \mathbf{O} && \text{(OS-level Constants)} \\
x &\in \mathbf{V} && \text{(Variables)} \\
e &\in \mathbf{E} && \text{(Expressions)} \\
\tau &\in \mathbf{T} && \text{(Types)} \\
\sigma &\in \mathbf{M} && \text{(Substitutions)}
\end{aligned}
$$

Figure 4.1: Syntactic categories of the language.

The categories can be described in the following way:

N (Numerals) Numerals, which are representations of natural numbers.

B (Booleans) Booleans, which represent the truth values: ‘true‘ and ‘false‘.

S (String) Strings, which are sequences of characters used to represent text. Examples include ”hello”, ”world”, and ”Christoffer”.

P (Filepath) File paths, which are strings representing the locations of files within a filesystem. File paths can be absolute, starting from the root directory, or relative, starting from the current directory.

V (Variables) Variables, which are placeholders for values that can change during program execution. Variables are used to store data that can be modified or retrieved later.

O (OS-level Constants) OS-level constants, which are specific to the operating system environment and remain constant. Examples include environment variables and predefined system paths.

E (Expressions) Expressions, which are combinations of variables, constants, and operators that evaluate to a value. The formation rules for **E** is described in Figure 4.2.

$$e ::= n \mid b \mid s \mid p \mid o \mid x \mid \lambda x : \tau.e \mid e[\sigma] \mid e_1 \circ e_2 \mid \lfloor e \rfloor \mid \lceil e \rceil$$

Figure 4.2: Formation rules for **E**.

T (Types) Types, which classify data into categories such as integer, boolean, string, etc. Types help define the kind of operations that can be performed on the data and how the data is stored. This category can be seen in Figure 4.3.

$$\tau = \mathit{string} \mid \mathit{unit} \mid \mathit{filepath} \mid \mathit{num} \mid \mathit{bool} \mid \lfloor \tau \rfloor \mid \tau_1 \to \tau_2 \mid \tau_1 \circ \tau_2$$

Figure 4.3: The types in the λ_{sh}.

M (Substitution) In this language we use explicit substitution and for that reason we need **M**. The formation rules for **M** is described in Figure 4.4.

$$\sigma ::= \texttt{id} \mid (e/x) \cdot \sigma$$

Figure 4.4: The formation rules for substitutions.

In conclusion, the syntactic categories and formation rules of λ_{sh} provide a foundation for representing and manipulating various types of data and operations within the language. By extending the traditional λ-calculus with additional categories and rules, λ_{sh} is equipped to handle the concepts found in shell scripting.

4.3 Semantics

The semantics of λ_{sh} provides a formal framework for understanding the behaviour and execution of programs written in the language. This section describes the small-step operational semantics, explicit substitutions, values, constant evaluation, and the shell state. By defining these aspects, we can precisely describe how λ_{sh} programs are evaluated and executed in a small-step semantics, ensuring correctness and predictability in their behaviour.

4.3.1 Small-step semantics

The semantics of λ_{sh} is defined through a small-step operational semantics. This approach specifies how individual computational steps are performed, allowing us to describe the execution of programs as a sequence of small, atomic transitions. Each step represents a single reduction or evaluation rule applied to a part of the program, gradually transforming it until a final value is reached. In small-step semantics, the execution of an expression e proceeds through a series of transitions:

$$e \to e_1 \to e_2 \to \cdots \to v$$

where each e_i is an intermediate expression and v is the final value. By using small-step semantics, we can formally reason about the correctness and properties of λ_{sh} programs, and we will use it later in order to prove the soundness property.

4.3.2 Explicit substitutions

In λ_{sh}, explicit substitutions are the mechanism for handling variable bindings. This approach incorporates bindings directly into the syntax, which is crucial for modelling the language using small-step semantics. Explicit substitutions allow us to accurately maintain the correct environment throughout partial evaluation steps. This approach is borrowed/inspired by Abadi et al[1].

Explicit substitution involves systematically replacing variables in an expression with their corresponding values as defined by a substitution mapping. This method is essential for implementing variables and scopes within the language. Because λ_{sh} is described using small-step semantics, it is necessary to store variable bindings for each partial transition. The substitution mapping, denoted by σ, is a finite mapping from variables to values and is an integral part of the syntax. A substitution mapping is represented syntactically as shown in Figure 4.4.

This systematic approach to substitutions ensures that variable bindings are consistently and correctly managed throughout the evaluation process in λ_{sh}.

4.3.3 Values

Values in λ_{sh} are categorised as shown in Figure 4.5.

$$\mathbb{n} \in \mathbb{N}$$
$$\mathbb{b} \in \mathbb{B}$$
$$\mathbb{s} \in \mathbb{S}$$
$$\mathbb{p} \in \mathbb{P}$$
$$\mathbb{c} \in \mathbb{C}$$
$$\mathbb{q} \in \mathbb{Q}$$
$$\mathbb{u} \in \mathbb{U}$$

$$\mathbb{V} = \mathbb{N} \cup \mathbb{B} \cup \mathbb{S} \cup \mathbb{P} \cup \mathbb{C} \cup \mathbb{Q} \cup \mathbb{U}$$

Figure 4.5: Definition of the value categories in λ_{sh}.

The definitions of the categories are the following:

$\mathbb{N}$ The set of natural numbers. These include all positive integers starting from 1, 2, 3, and so on. Natural numbers are typically used for counting and ordering.

$\mathbb{B}$ The set of boolean values. This set contains exactly two elements: `true` and `false`. Boolean values are used in logic operations and conditions.

$\mathbb{S}$ The set of strings. Strings are sequences of characters and are used to represent text. Examples include "hello", "world", and "123".

$\mathbb{P}$ The set of file paths. File paths are strings that represent the locations of files within a filesystem. They may be absolute (starting from the root directory) or relative (relative to some other directory).

$\mathbb{C}$ The set of closures. Closures are functions along with the referencing environment for the non-local variables of that function. They are used in language to encapsulate functionality along with the context in which it operates.

$\mathbb{Q}$ The set of quoted values. A quote is just an expression which execution has been suspended. So the actual value of a quoted expression is just **E**. Therefore, a quoted expression can be seen as a function without parameters.

$\mathbb{U}$ The set of the unit value. In the semantics we use $\mathbb{u}$ to represent the unit value.

$\mathbb{V}$ The set of all values. These include all values from all other categories, and are used when we need to say that we have some value, but not any specific value from a category.

4.3.4 Constant evaluation

We define a function γ in Figure 4.6 that evaluates values from the syntactic categories to the values of the value categories. This constant evaluation function is used to define our value axioms in the semantics.

$$\gamma(n) = \text{value of type } \mathbb{N} \quad \text{(Numerals)}$$
$$\gamma(b) = \text{value of type } \mathbb{B} \quad \text{(Booleans)}$$
$$\gamma(s) = \text{value of type } \mathbb{S} \quad \text{(Strings)}$$
$$\gamma(p) = \text{value of type } \mathbb{P} \quad \text{(Filepaths)}$$

Figure 4.6: Caption

4.3.5 State

In λ_{sh}, the shell state is a component that encapsulates the current environment of the shell, including the current working directory and the file system. The shell state is denoted by δ and is a tuple consisting of the current working directory and the file system. This section defines the shell state and its role in the operational semantics of λ_{sh}.

The shell state δ is represented as a tuple of the current working directory $\mathbb{p}$ and the file system `fs`. It belongs to the category of states, denoted by Δ.

$$\delta \in \Delta \quad \text{where} \quad \delta = (\mathbb{p}, \texttt{fs})$$

The current working directory, represented by $\mathbb{P}$, is a file path that indicates the directory in which the shell is currently operating. It is essential for resolving relative file paths and executing commands within the correct context.

File System Representation

To represent an element in the file system, we define file values, $FVal$:

$$FVal = \mathbb{P} \cup File \cup Dir$$

Where:

- $\mathbb{P}$ represents aliases within the file system.
- $File$ (denoted as Σ^*) represents files.
- Dir represents directories.

Filesystem Tree A filesystem tree is described as a partial mapping from paths to sets of file values:

$$Dir = \mathbb{P} \rightharpoonup \mathcal{P}(FVal)$$

Filesystem Lookup A path p within the operating system can be described by the function fs, which performs a lookup in the filesystem tree:

$$fs : \mathbb{P} \rightharpoonup FVal$$
$$fs(p) = Dir(p)$$

Operations on the Filesystem We define several important operations on the filesystem, including insertion, deletion, and referencing the current working directory.

Insertion Insertion of a file value $fval$ at a path p is represented by extending the filesystem fs:

$$fs[p \mapsto fval]$$

This operation adds $fval$ to the set of values associated with the path p.

Deletion Deletion of a file at a path p is represented by removing the value at the specified path:

$$fs[p \mapsto \texttt{undef}]$$

This operation sets the path p as undefined in the filesystem.

Current Working Directory To reference the current subtree or current working directory, we denote it as *cwd*. The current working directory is a path value in the filesystem.

In conclusion, the semantic tools of λ_{sh} are provided such that we can define semantics for understanding how programs are executed. The small-step operational semantics offer a step-by-step approach to program evaluation, while explicit substitutions handle variable bindings effectively. The categorisation of values and the constant evaluation function ensure that data is manipulated consistently and correctly. Finally, the definition of the shell state encapsulates the current environment of the shell, allowing for accurate modelling of the execution context. Together, these components form a comprehensive semantic framework.

4.4 Type system

Having established the operational semantics of λ_{sh}, we now turn our attention to the type system. The type system serves as a formal framework that categorise expressions into different types, ensuring that operations are performed on compatible types and thus preventing type errors during evaluation.

A well-defined type system is crucial for guaranteeing the correctness of programs written in λ_{sh}. It provides the foundation for proving important properties such as type safety, which ensures that well-typed programs do not cause runtime type errors. In the following sections, we will define the syntactic categories of the type system, introduce the rules for type checking, and explore the typing rules for various constructs in λ_{sh}.

We will demonstrate in Chapter 5 that λ_{sh} type system is sound, which means that well-typed programs are guaranteed to evaluate correctly according to the operational semantics we have defined.

4.4.1 The typing context

In the λ_{sh} language, the typing context Γ plays the role of ensuring that variables and expressions are used consistently according to their types throughout a script. The typing context is a formal construct that maps variables to their respective types, providing the necessary information for type checking and type inference within the language.

$$\Gamma : \textbf{Var} \rightharpoonup \textbf{Types}$$

The typing context Γ is defined as a finite set of assumptions about the types of variables. Formally, it can be expressed as:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

where Γ is either the empty context $\emptyset$ or an extension of an existing context Γ with a new variable x and its associated type τ.

The primary function of the typing context Γ is to provide a mapping from variables to their types, which is essential for the type checking process. When an expression is evaluated, the typing context ensures that each variable used within the expression has a well-defined type, thereby preventing type errors.

Variable Typing For a variable x, the typing context Γ allows us to determine its type by looking it up in the context. This can be expressed as:

$$[\text{T-VAR}] \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

This rule states that if a variable x has a type τ in the typing context Γ, then x is well-typed with type τ under Γ.

Extending the Typing Context As a script is type-checked, the typing context Γ is extended to include new variable bindings. For example, when a new variable is introduced in a lambda-binding, the context is updated to include the type of the new variable. This is crucial for maintaining the integrity of the type system throughout the script. By maintaining a consistent mapping of variables to types, the language can detect and prevent type errors at compile time, thus ensuring that well-typed programs do not produce type errors during execution. This formal mechanism provides a foundation for building reliable and robust scripts.

In summary, the typing context Γ in λ_{sh} is an essential component of the type system, providing a structured way to manage and enforce the types of variables and expressions.

4.4.2 Type rules

In λ_{sh}, type rules are used to define how expressions and commands are assigned types within the language. These rules are for ensuring that programs are type-safe, meaning that they do not produce type errors during execution. Type rules are written in a formal notation that specifies the conditions under which a particular expression or command has a given type.

A type rule in λ_{sh} is written in the form of an inference rule, as described in Chapter 2:

$$\frac{\Gamma \vdash \text{Premises}}{\Gamma \vdash \text{Conclusion}}$$

where the premises are conditions that must be satisfied for the conclusion to hold in the given type context. The conclusion states the type of an expression or command given the premises. The type system is then defined as a collection of these rules.

4.5 Transition system

Before we can define the small-step semantic rules, we must first define the transition system.

The transition system for evaluating λ_{sh} is defined as a triple $(C, \rightarrow, T)$, where C is the configuration space, $\rightarrow$ is the transition relation, and T is the set of terminal configurations.

The configurations C and terminal configurations T are defined as shown in Figure 4.7.

$$C = (\mathbf{E} \times \Delta) \cup (\mathbb{V} \times \Delta)$$
$$T = \mathbb{V} \times \Delta$$

Figure 4.7: Transition system configurations.

Transitions in the system are written in the form $\langle e, \delta \rangle \to \langle e', \delta' \rangle$, indicating that an expression e with the state δ transitions to a new expression e' with an updated state δ'.

Now that the basis for the transition system has been defined, we can go into detail with specific language features.

4.6 Structural congruence

Structural congruence allows us to consider two syntactically different expressions as equivalent if they can be transformed into each other through a series of structural rules, defined through equivalences. The rules of structural congruence provide a way to formalise this equivalence. These rules enable us to manipulate expressions to reveal their underlying structure and equivalence, rewriting them into different forms, which is crucial for simplifying complex expressions and proving properties about systems.

Figure 4.8 present the fundamental rule for structural congruence in λ_{sh}:

$$\text{[STRUCT]} \qquad \frac{e \equiv e_1 \quad \langle e_1, \delta \rangle \to \langle e_1', \delta' \rangle \quad e' \equiv e_1'}{\langle e, \delta \rangle \to \langle e', \delta' \rangle}$$

Figure 4.8: Rule for faciliating the use of structural congruence

The [STRUCT] rule is crucial for maintaining the consistency of the evaluation process. It states that if an expression e is structurally congruent to another expression e_1, and e_1 can evaluate to e_1' with a corresponding state evaluation from δ to δ', then e can evaluate to an expression e' that is structurally congruent to e_1' with the same state transition.

4.7 Values

Values are the basic units of data that the language manipulates, including numerals, booleans, strings, file paths, unit values, and variables. We begin by detailing the small-step semantics, which describe how these values are computed step-by-step, followed by the type rules that ensure values are used consistently according to their types. First we need to define the structural equivalences that are valid for the simple value formations. Those can be seen in Figure 4.9.

$$\text{[S-NUM]} \quad n[\sigma] \equiv n$$

$$\text{[S-BOOL]} \quad b[\sigma] \equiv b$$

$$\text{[S-STRING]} \quad s[\sigma] \equiv s$$

$$\text{[S-FILEPATH]} \quad p[\sigma] \equiv p$$

$$\text{[S-UNIT]} \quad \texttt{unit}[\sigma] \equiv \texttt{unit}$$

Figure 4.9: The equivalences for values formations.

Figure 4.9 presents several equivalences that describe how different types of values remain unchanged when a substitution is applied to them. These equivalences ensure that the fundamental values in our formal system are stable under substitution, preserving their identity and integrity.

The [S-NUM] equivalence states that when a substitution σ is applied to a numeral n, the numeral remains unchanged. Formally, this is expressed as $n[\sigma] \equiv n$. This rule indicates that numeric values are constants and do not vary with substitutions. This is fundamentally the same for all the other rules. Now, the actual reduction rules for the value formations can be seen in Figure 4.10.

$$\text{[NUM]} \quad \frac{}{\langle n, \delta\rangle \to \langle \mathbb{n}, \delta\rangle} \text{ if } \gamma(n) = \mathbb{n}$$

$$\text{[BOOL]} \quad \frac{}{\langle b, \delta\rangle \to \langle \mathbb{b}, \delta\rangle} \text{ if } \gamma(b) = \mathbb{b}$$

$$\text{[STRING]} \quad \frac{}{\langle s, \delta\rangle \to \langle \mathbb{s}, \delta\rangle} \text{ if } \gamma(s) = \mathbb{s}$$

$$\text{[FILEPATH]} \quad \frac{}{\langle p, \delta\rangle \to \langle \mathbb{p}, \delta\rangle} \text{ if } \gamma(p) = \mathbb{p}$$

$$\text{[UNIT]} \quad \frac{}{\langle unit, \delta\rangle \to \langle \mathbb{u}, \delta\rangle}$$

$$\text{[VAR]} \quad \frac{}{\langle x\sigma, \delta\rangle \to \langle \mathbb{v}, \delta\rangle} \text{ if } x \mapsto \mathbb{v} \in \sigma$$

Figure 4.10: Small-step semantics for basic syntactic categories.

The semantic rules in Figure 4.10 describe the evaluation process for various basic expressions in a small-step operational semantics. Each rule shows a single evaluation step. The [NUM] rule states that if a numeric literal n is constantly evaluated to the value $\mathbb{n}$ in the interpretation function γ, then $\langle n, \delta\rangle$ evaluates to $\langle \mathbb{n}, \delta\rangle$. This is roughly the same for the rules [BOOL], [STRING], [FILEPATH], and [UNIT]. The [VAR] rule states that if a variable x maps to the value $\mathbb{v}$ in the substitution environment σ, then $\langle x\sigma, \delta\rangle$ evaluates to $\langle \mathbb{v}, \delta\rangle$. These rules show how numbers, booleans, strings, file paths, unit values, and variables are evaluated step-by-step, ensuring each expression reduces to its corresponding value while keeping the state δ unchanged.

$$\text{[T-NUM]} \quad \frac{}{\Gamma \vdash n : \texttt{num}}$$

$$\text{[T-BOOL]} \quad \frac{}{\Gamma \vdash b : \texttt{bool}}$$

$$\text{[T-STRING]} \quad \frac{}{\Gamma \vdash s : \texttt{string}}$$

$$\text{[T-FILEPATH]} \quad \frac{}{\Gamma \vdash p : \texttt{filepath}}$$

$$\text{[T-UNIT]} \quad \frac{}{\Gamma \vdash u : \texttt{unit}}$$

$$\text{[T-VAR]} \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Figure 4.11: Type rules for basic syntactic categories.

In Figure 4.11, we have the type rules for literals and variables. First, the rule [T-NUM] specifies that a numeral n is always of type `num`. Similarly, the rule [T-BOOL] specifies that a boolean b is always of type `bool`, and the rule [T-STRING] specifies that a string s is always of type `string`. These rules ensure that in any typing context Γ, numerals, booleans, and strings are recognised with their correct types without any ambiguity.The rule [T-FILEPATH] specifies that a filepath p is always of type `filepath`, and the rule [T-UNIT] specifies that a unit value u is always of type `unit`. These rules ensure that file paths and unit values are also correctly typed in any context. The rule [T-VAR] is slightly different, as it deals with variables rather than literals. It ensures that a variable x has the type τ if the typing context Γ maps x to τ. This means that if the context Γ associates the variable x with type τ, then x has type τ. This rule is ensures that the type checker can verify that bindings are used according to their declared types.

4.8 Substitution, abstraction, and application

In λ_{sh}, variable substitution, abstraction and application are fundamental constructs. This section explores the small-step semantics and type rules for these constructs and how they are applied and used.

Figure 4.12 illustrates the equivalence rules that govern variable substitution, lambda abstraction, and function application in our formal system. These rules define how expressions can be transformed or simplified while preserving their meaning. Below is a detailed explanation of each rule presented in the figure.

[BETA]	$(\lambda x.e_1)e_2 \equiv e_1[(e_2/x)]$
[VAR-1]	$x[(e/x) \cdot s] \equiv e$
[VAR-2]	$x[(e/y) \cdot s] \equiv x[s]$
[VAR-3]	$x[\mathtt{id}] \equiv x$
[APP]	$(e_1\ e_2)[\sigma] \equiv (e_1[\sigma])(e_2[\sigma])$
[ABS]	$(\lambda x.e)[s] \equiv \lambda y.(e[(y/x) \cdot s])$
[APP-1]	$\dfrac{e_1\ e \equiv e_2\ e}{e_1 \equiv e_2}$
[APP-2]	$\dfrac{e\ e_1 \equiv e\ e_2}{e_1 \equiv e_2}$
[SYMM]	$\dfrac{e' \equiv e}{e \equiv e'}$
[TRANS]	$\dfrac{e \equiv e_1 \quad e_1 \equiv e'}{e \equiv e'}$

Figure 4.12: Equivalence rules variable substitution, abstraction, and application.

First, the [BETA] rule describes the process of beta reduction, which is the application of a lambda abstraction to an argument. Specifically, the expression $(\lambda x.e_1)e_2$ is equivalent to $e_1[(e_2/x)]$, meaning that e_2 is substituted for x in e_1. This rule is fundamental for evaluating function applications. Next, we have three rules governing variable substitution. The [VAR-1] rule states that substituting e for x in x yields e. Formally, this is written as $x[(e/x) \cdot s] \equiv e$. The [VAR-2] rule addresses the case where x and y are different variables; substituting e for y in x leaves x unchanged, hence $x[(e/y) \cdot s] \equiv x[s]$. Lastly, the [VAR-3] rule indicates that applying the identity substitution to a variable x results in x itself, expressed as $x[\mathtt{id}] \equiv x$. The [APP] rule focuses on function application and substitution. It asserts that applying a substitution σ to the application of two expressions e_1 and e_2 is equivalent to applying the substitution to each expression individually and then applying the resulting expressions together. For lambda abstractions, the [ABS] rule explains how to apply a substitution s to a lambda abstraction. The expression $(\lambda x.e)[s]$ is equivalent to $\lambda y.(e[(y/x) \cdot s])$, where y is a fresh variable that does not occur in e or s. This rule helps maintain the correct scope and binding of variables during substitution. Finally, the figure includes two rules for application equivalence. The [APP-1] rule states that if two expressions e_1 and e_2 are equivalent, then applying another expression e to both results in equivalent applications: $\dfrac{e_1\ e \equiv e_2\ e}{e_1 \equiv e_2}$. Similarly, the [APP-2] rule asserts that if e_1 and e_2 are equivalent, then applying them to another expression e results in equivalent applications: $\dfrac{e\ e_1 \equiv e\ e_2}{e_1 \equiv e_2}$. We also have reduction rules for applications that mimic running a external commands in a shell. They are given in Figure 4.13.

$$\text{[APP-LEFT]} \quad \frac{\langle e_1, \delta\rangle \to \langle e_1', \delta'\rangle}{\langle e_1\ e_2, \delta\rangle \to \langle e_1'\ e_2, \delta'\rangle}$$

$$\text{[APP-RIGHT]} \quad \frac{\langle e_2, \delta\rangle \to \langle e_2', \delta'\rangle}{\langle \mathbb{v}_1\ e_2, \delta\rangle \to \langle \mathbb{v}_1\ e_2', \delta'\rangle}$$

$$\text{[APP-FILEPATH]} \quad \frac{}{\langle \mathbb{p}\ \mathbb{s}, \delta\rangle \to \langle \mathbb{u}, \delta\rangle}$$

$$\text{[APP-STRING]} \quad \frac{}{\langle \mathbb{s}\ \mathbb{s}, \delta\rangle \to \langle \mathbb{u}, \delta\rangle}$$

Figure 4.13: Small-step semantics for application.

First, the [APP-LEFT] rule describes the evaluation of the left side of an application. Specifically, if the expression $\langle e_1, \delta\rangle$ transitions to $\langle e_1', \delta'\rangle$, then the application $\langle e_1\ e_2, \delta\rangle$ transitions to $\langle e_1'\ e_2, \delta'\rangle$. This rule allows the evaluation to proceed by focusing on the leftmost part of the application first. Next, the [APP-RIGHT] rule handles the evaluation of the right side of an application when the left side is already a value. If the expression $\langle e_2, \delta\rangle$ transitions to $\langle e_2', \delta'\rangle$, then the application $\langle \mathbb{v}_1\ e_2, \delta\rangle$ transitions to $\langle \mathbb{v}_1\ e_2', \delta'\rangle$. Here, $\mathbb{v}_1$ is a value, so the focus shifts to evaluating the right side of the application. The [APP-FILEPATH] rule describes a specific application where a file path is applied to a string. The application $\langle \mathbb{p}\ \mathbb{s}, \delta\rangle$ transitions to $\langle \mathbb{u}, \delta\rangle$, where $\mathbb{u}$ represents the unit value. This rule captures the semantics of applying a file path to a string, resulting in a unit type. Finally, the [APP-STRING] rule handles the application of two string values. The application $\langle \mathbb{s}\ \mathbb{s}, \delta\rangle$ transitions to $\langle \mathbb{u}, \delta\rangle$, where $\mathbb{u}$ is the unit value. This rule defines the evaluation of applying one string to another, resulting in a unit type.

Together, these rules define how abstractions work, variable substitutions, how closures are applied, and how the shell specific operations work. They ensure a systematic method for handling function-related expressions within the overall semantics. We also have the type rules for variable substitutions, abstractions and applications as described in Figure 4.14.

$$\text{[T-ABS]} \quad \frac{\Gamma[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.e) : \tau \to \tau'}$$

$$\text{[T-APP]} \quad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \tau'}$$

$$\text{[T-CLOS]} \quad \frac{\Gamma \vdash \sigma \triangleright \Gamma' \quad \Gamma' \vdash e : \tau}{\Gamma \vdash e[\sigma] : \tau}$$

$$\text{[T-ID]} \quad \frac{}{\Gamma \vdash \texttt{id} \triangleright \Gamma}$$

$$\text{[T-CONS]} \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \sigma \triangleright \Gamma'}{\Gamma \vdash (e/x) \cdot \sigma \triangleright \Gamma'[x \mapsto \tau]}$$

$$\text{[T-APP-FILEPATH]} \quad \frac{\Gamma \vdash e_1 : \texttt{filepath} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1\ e_2 : \texttt{unit}}$$

$$\text{[T-APP-STRING]} \quad \frac{\Gamma \vdash e_1 : \texttt{string} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1\, e_2 : \texttt{unit}}$$

Figure 4.14: Type rules for abstraction and application.

First, the [T-ABS] rule handles lambda abstractions. It states that if under the context Γ, extending Γ with x mapped to τ results in e having type τ', then the lambda abstraction $\lambda x : \tau.e$ has the function type $\tau \to \tau'$. Next, the [T-APP] rule defines the type for function application. If under the context Γ, e_1 has type $\tau \to \tau'$ and e_2 has type τ, then the application $e_1\, e_2$ has type τ'. The [T-CLOS] rule deals with expressions under substitutions. If under the context Γ, a substitution σ results in a new context Γ' such that e has type τ in Γ', then $e[\sigma]$ has type τ in Γ.

The $\triangleright$ operator in the type rules serves to manage and track substitutions within typing contexts. The $\triangleright$ operator specifically facilitates the application of substitutions to typing contexts, ensuring that the resulting contexts accurately reflect the transformations applied to expressions. In the type rules, the $\triangleright$ operator is used to denote the relationship between an original context Γ and a new context Γ' after applying a substitution σ. The [T-ID] rule states that the identity substitution `id` maps a context Γ to itself. The [T-CONS] rule explains how to type a substitution that extends an environment. If under the context Γ, e has type τ and the substitution σ maps Γ to Γ', then the extended substitution $(e/x) \cdot \sigma$ maps Γ to Γ' extended with x mapped to τ.

The [T-APP-FILEPATH] rule specifies the type for applying a filepath to a string. If under the context Γ, e_1 has type `filepath` and e_2 has type `string`, then the application $e_1\, e_2$ has type `unit`. Similarly, the [T-APP-STRING] rule defines the type for applying one string to another. If under the context Γ, both e_1 and e_2 have type `string`, then the application $e_1\, e_2$ has type `unit`. The semantics and type rules for variable substitution, abstraction and application in λ_{sh} provide a systematic approach to evaluating and typing function-related expressions. These rules ensure that substitutions are correctly typed and evaluated, lambda abstractions are correctly formed, closures are applied consistently, and shell-specific operations are handled appropriately.

4.9 Shell operations

λ_{sh} includes core shell commands and their semantics to interact with the file system and the current working directory. The small-step semantics for these operations are defined in Figure 4.15.

$$[\text{CWD}] \quad \frac{}{\langle \texttt{cwd}, \delta \rangle \to \langle \mathbb{p}, \delta \rangle} \text{ where } \delta = (\mathbb{p}, \texttt{fs})$$

$$[\text{CD-1}] \quad \frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{cd } e, \delta \rangle \to \langle \texttt{cd e'}, \delta \rangle}$$

$$[\text{CD-2}] \quad \frac{}{\langle \texttt{cd } \mathbb{p}, \delta \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ where } \delta' = (\mathbb{p}, \texttt{fs})$$

$$[\text{TOUCH-1}] \quad \frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{touch } e, \delta \rangle \to \langle \texttt{touch } e', \delta' \rangle}$$

$$[\text{TOUCH-2}] \quad \frac{}{\langle \texttt{touch } \mathbb{p}, (\mathbb{p}', \texttt{fs}) \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ if } \delta' = (\mathbb{p}', \texttt{fs}[\mathbb{p} \mapsto fval])$$

$$[\text{MKDIR-1}] \quad \frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{mkdir } e, \delta \rangle \to \langle \texttt{mkdir } e', \delta' \rangle}$$

$$[\text{MKDIR-2}] \quad \frac{}{\langle \texttt{mkdir } \mathbb{p}, (\mathbb{p}', \texttt{fs}) \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ if } \delta' = (\mathbb{p}', \texttt{fs}[\mathbb{p} \mapsto fval])$$

$$[\text{RM-1}] \quad \frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{mkdir } e, \delta \rangle \to \langle \texttt{mkdir } e', \delta' \rangle}$$

$$[\text{RM-2}] \quad \frac{}{\langle \texttt{rm } \mathbb{p}, (\mathbb{p}', \texttt{fs}) \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ if } \delta' = (\mathbb{p}', \texttt{fs}[\mathbb{p} \mapsto \texttt{undef}])$$

Figure 4.15: Small-step semantics for built in shell functionality.

The semantic rules in Figure 4.15 define the evaluation process for core shell commands within the λ_{sh} language. Each rule describes how specific commands interact with the current state δ. The [CWD] rule handles the `cwd` command, which retrieves the current working directory. Given the state $\delta = (\mathbb{p}, \texttt{fs})$, the command evaluates to the current working directory $\mathbb{p}$ without changing the state, denoted as $\langle \texttt{cwd}, \delta \rangle \to \langle \mathbb{p}, \delta \rangle$.

The [CD-1] and [CD-2] rules address the evaluation of the *cd* command, which changes the current working directory. The [CD-1] rule manages the initial evaluation step, taking the command argument e and progressing it to e'. This is shown as $\langle \texttt{cd } e, \delta \rangle \to \langle \texttt{cd e'}, \delta \rangle$ if $\langle e, \delta \rangle \to \langle e', \delta' \rangle$. The [CD-2] rule finalises the evaluation when the argument is fully evaluated to a value $\mathbb{p}$, updating the current working directory and leaving the file system unchanged, formally expressed as $\langle \texttt{cd } \mathbb{p}, \delta \rangle \to \langle \mathbb{u}, \delta' \rangle$, where $\delta' = (\mathbb{p}, \texttt{fs})$.

The [TOUCH-1] and [TOUCH-2] rules describe the behaviour of the `touch` command, which creates a new file. The [TOUCH-1] rule handles the partial evaluation of the command argument, evaluating from e to e', shown as $\langle \texttt{touch } e, \delta \rangle \to \langle \texttt{touch } e', \delta \rangle$ if $\langle e, \delta \rangle \to \langle e', \delta' \rangle$. The [TOUCH-2] rule completes the evaluation when the argument is a fully evaluated path $\mathbb{p}$, creating a new file in the file system. The [MKDIR-1] and [MKDIR-2] rules are similar but pertain to the 'mkdir' command, which creates a new directory. The initial evaluation step is managed by [MKDIR-1], evaluating from e to e', indicated as $\langle \texttt{mkdir } e, \delta \rangle \to \langle \texttt{mkdir } e', \delta \rangle$ if $\langle e, \delta \rangle \to \langle e', \delta' \rangle$. The final evaluation step, described by [MKDIR-2], creates a new directory at the evaluated path $\mathbb{p}$. Finally, the [RM-1] and [RM-2] rules describe the behaviour of the 'rm' command, which removes a file or directory. The [RM-1] rule handles the initial evaluation of the command argument, evaluating from e to e', shown as $\langle \texttt{rm } e, \delta \rangle \to \langle \texttt{rm } e', \delta \rangle$ if $\langle e, \delta \rangle \to \langle e', \delta' \rangle$. The [RM-2] rule completes the evaluation by removing the file or directory at the evaluated path $\mathbb{p}$. These rules collectively ensure that core shell commands in λ_{sh} are evaluated correctly, reflecting their intended

behaviour in modifying the shell state and file system.

$$\text{[T-CWD]} \quad \frac{}{\Gamma \vdash \texttt{cwd} : \texttt{filepath}}$$

$$\text{[T-CD]} \quad \frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{cd}\ e : \texttt{unit}}$$

$$\text{[T-TOUCH]} \quad \frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{touch}\ e : \texttt{unit}}$$

$$\text{[T-MKDIR]} \quad \frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{mkdir}\ e : \texttt{unit}}$$

$$\text{[T-RM]} \quad \frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{rm}\ e : \texttt{unit}}$$

Figure 4.16: Type rules for built-in functionality.

In Figure 4.16, we have the type rules for shell functionality. The rule [T-CWD] states that the `cwd` command, which retrieves the current working directory, has the type `filepath`. This ensures that whenever `cwd` is used, it is recognized as a file path.

The rule [T-CD] ensures that the `cd` command, used to change the current working directory, is correctly typed. The command takes an argument e of type `filepath` and results in a value of type `unit`. Similarly, the rule [T-TOUCH] verifies the typing for the `touch` command, which creates a new file. The command takes an argument e of type `filepath` and returns a value of type `unit`. The rule [T-MKDIR] specifies the typing for the `mkdir` command, which creates a new directory. Like `touch`, the `mkdir` command takes an argument e of type `filepath` and returns a value of type `unit`.

Finally, the rule [T-RM] ensures that the `rm` command, used to remove files or directories, is correctly typed. The command takes an argument e of type `filepath` and results in a value of type `unit`.

4.10 Quoting

Quoting in λ_{sh} is a mechanism used to delay the evaluation of expressions, allowing them to be treated as data rather than executable code. This feature is particularly useful for string interpolation, constructing complex commands, and manipulating code as data within the language. In many shell languages, we have the idea of building commands, which are code that need to include variables and expressions without immediately evaluating them. In λ_{sh}, quoting serves a similar purpose but extends its utility to managing the evaluation context of expressions more broadly. Quoting is denoted by the operator $\lfloor \cdot \rfloor$. When an expression e is quoted as $\lfloor e \rfloor$, its evaluation is suspended, and it is treated as a literal value rather than being executed immediately. This allows the expression to be passed around, stored, or manipulated without triggering its evaluation. Unquoting, denoted by the operator $\lceil \cdot \rceil$, resumes the evaluation of a quoted expression. When $\lceil e \rceil$ is encountered, the expression inside the quotes is evaluated in the current context. Now consider the following example where quoting is used to construct a command:

$$\text{gcc} = \lfloor \text{"gcc -ansi -Wall"} \rfloor$$

Here, gcc is a macro which calls the GCC compiler. We want to use this macro to compile files in our

operating system, we use the command by:

$$\text{cmd} = gcc \circ (cwd \circ "first\_file.c")$$

To execute the command, the expression is unquoted:

$$\lceil cmd \rceil$$

Unquoting cmd evaluates the expression $gcc \circ (cwd \circ "first\_file.c")$ under the current environment. A quoted value acts like a macro, waiting to be used until it is unquoted. The goal of this mechanism is to introduce a powerful tool to construct programs by string interpolation. The reduction rules for unquoting/quoting is defined by structural congruence and the [STRUCT] rule presented in section 4.6.

$$\text{[T-QUOTE]} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lfloor e \rfloor : \lfloor \tau \rfloor}$$

$$\text{[T-UNQUOTE]} \qquad \frac{\Gamma \vdash e : \lfloor \tau \rfloor}{\Gamma \vdash \lceil e \rceil : \tau}$$

Figure 4.17: Type rules for quoting.

In Figure 4.17, we have the type rules for quotation. First, the rule [T-QUOTE] specifies how an expression e is handled when it is quoted. Quoting an expression means treating it as a syntactic entity rather than evaluating it immediately. This rule means that if an expression e has type τ in the typing context Γ, then the quoted expression $\lfloor e \rfloor$ has the type $\lfloor \tau \rfloor$ in the same context. Quoting preserves the type information of the expression while marking it as quoted.

The rule [T-UNQUOTE] specifies how an expression e is handled when it is unquoted. Unquoting an expression means evaluating it within the current context. This rule means that if a quoted expression e has the type $\lfloor \tau \rfloor$ in the typing context Γ, then the unquoted expression $\lceil e \rceil$ has the type τ in the same context. Unquoting effectively removes the quotation, allowing the expression to be evaluated with its original type.

4.10.1 Quoting laws

This subsection will expand upon the quoting laws presented in section 4.6 with some rules for type equivalence. Some quoted expressions need additional rules to ensure structural congruence. Quoting in λ_{sh} provides a powerful mechanism for managing when and how expressions are evaluated, enabling more flexible and dynamic shell scripting capabilities, while still maintaining the properties desired for static analysis. To ensure consistent behaviour, λ_{sh} defines several laws governing quoting and unquoting:

$$[\text{REFL}] \qquad e \equiv e$$

$$[\text{INV}] \qquad \lceil \lfloor e \rfloor \rceil \equiv e$$

$$[\text{IDEM}] \qquad \lfloor \lfloor e \rfloor \rfloor \equiv \lfloor e \rfloor$$

$$[\text{QCON}] \qquad \lfloor e_1 \rfloor \circ \lfloor e_2 \rfloor \equiv \lfloor e_1 \circ e_2 \rfloor$$

Figure 4.18: Rules for structural congruence of quoted expressions

These laws ensure that quoting and unquoting operations behave predictably and consistently within the language. The first law states that $\lceil e \rceil$ and $\lfloor e \rfloor$ and are each others inverse as we can see in the [INV] rule, the quoting mechanism is also idempotent (rule [IDEM]), nested quotes have no meaning. The last rule [QCON] is for concatination of quoted expressions, the [QCON] rule states that concatinating quoted expressions should be still be quoted. However, the [QCON] is not always correct. We have to introduce laws for type equivalence to guarantee correctness of quoting.

$$[\text{T-IDEM}] \qquad \lfloor \lfloor \tau \rfloor \rfloor = \lfloor \tau \rfloor$$

$$[\text{T-CONSTR}] \qquad \mathit{string} \circ \mathit{string} = \mathit{string}$$

$$[\text{T-QSUB}] \qquad \lfloor \mathit{string} \rfloor = \mathit{string}$$

$$[\text{T-QCON}] \qquad \lfloor \tau_1 \rfloor \circ \lfloor \tau_2 \rfloor = \lfloor \tau_1 \circ \tau_2 \rfloor$$

Figure 4.19: Rules for type equivalence for quoted expressions

Now, the rule [T-IDEM] states that a quoted type is equal to the same type with nested quotes. This rule ensures that quoted types are idempotent. the rule *[T-CONSTR]* states that a concatination type that is only strings is just of type string. In section 4.11 we introduce subtyping to the type system because we want to be able to use quotes as strings, however for this to be sound later, we need to introduce a type equivalence rule for the subtyping. Finally, concatinating quoted types must follow the rule [QCON]

4.11 Subtyping

The general idea of the quoting construct is to pack base types into a quoted type $\lfloor \tau \rfloor$ and unquoting will unpack the basetype from the quoted expression. We need a mechanism which allows an expression of type S to be used as type T and this mechanism is called subtyping (subtype polymorphism). To describe this relationship, let $S <: T$ be the subtyping relation which states "S is a subtype of T" which is described by the $T\text{-}SUB$ inference rule. We define an axiom for the subtyping relation which described *string* as the maximum element, and any quoted type $\lfloor \tau \rfloor$ is a subtype of string.

$$\begin{aligned}
\texttt{bool} &<: \texttt{string} \\
\texttt{num} &<: \texttt{string} \\
\texttt{filepath} &<: \texttt{string} \\
\texttt{unit} &<: \texttt{string} \\
\lfloor \tau \rfloor &<: \texttt{string}
\end{aligned}$$

We formalise the subtyping relation by providing inference rules for deriving expressions presented in Figure 4.20. First we have that any subtyping relation need to be reflexive and transitive, this follows from the rule of subsumption. Subtyping abstractions require more work, we need to consider the *contravariants* and *covariants* of the subtyping relation [11].

Covariance is the variance described directly by the subtyping relation, if we have a function f of type $\tau_{11} \to \tau_{12}$, then under the subtyping relation, f has the type $\tau_{21} \to \tau_{22}$ where $\tau_{11} <: \tau_{21}$ and $\tau_{12} <: \tau_{22}$. We can observe this by the concatenation operator $\tau_{11} \circ \tau_{12} <: \tau_{21} \circ \tau_{22}$ [11].

Contravariance is the subtyping relation described by reverse subtyping relation (that is, the subtyping relation for base types is reversed for $\lfloor \tau \rfloor$). This is only present in the left-hand premise of the [T-ARROW] rule, where the parameter of the abstraction τ_{21} is safe in the context of the abstraction $\tau_{21} <: \tau_{11}$ [11].

$$\text{[T-SUB]} \quad \frac{\Gamma \vdash e : \tau_s \quad \tau_s <: \tau}{\Gamma \vdash e : \tau}$$

$$\text{[T-REFL]} \quad \frac{}{\tau <: \tau}$$

$$\text{[T-TRANS]} \quad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$\text{[T-ARROW]} \quad \frac{\tau_{21} <: \tau_{11} \quad \tau_{12} <: \tau_{22}}{\tau_{11} \to \tau_{12} <: \tau_{21} \to \tau_{22}}$$

Figure 4.20: Type rules for subtyping.

4.12 String operators

$$\text{[CONCAT-1]} \quad \frac{\langle e_1, \delta \rangle \to \langle e_1', \delta' \rangle}{\langle e_1 \circ e_2, \delta \rangle \to \langle e_1' \circ e_2, \delta' \rangle}$$

$$\text{[CONCAT-2]} \quad \frac{\langle e_2, \delta \rangle \to \langle e_2', \delta' \rangle}{\langle \mathbb{v} \circ e_2, \delta \rangle \to \langle \mathbb{v} \circ e_2', \delta' \rangle}$$

$$\text{[CONCAT-3]} \quad \frac{}{\langle \mathbb{v}_1 \circ \mathbb{v}_2, \delta \rangle \to \langle \mathbb{v}, \delta \rangle} \text{ if } \mathbb{v} = \mathbb{v}_1 \circ \mathbb{v}_2$$

Figure 4.21: Small-step semantics for concatenation.

The semantics for concatenation in Figure 4.21 describe how expressions involving the concatenation operator ($\circ$) are evaluated step-by-step. These rules ensure that concatenation operations are properly reduced until they reach their final form. The [CONCAT-1] rule handles the case where the left operand of the concatenation needs further evaluation. If the expression e_1 evaluates to e_1' with an updated state δ', then the concatenation $e_1 \circ e_2$ transitions to $e_1' \circ e_2$ with the same updated state δ'. This ensures that e_1 is fully evaluated before the concatenation proceeds. The [CONCAT-2] rule addresses the evaluation of the right operand when the left operand is already a value $\mathbb{v}$. If the expression e_2 evaluates to e_2' with an updated state δ', then the concatenation $\mathbb{v} \circ e_2$ transitions to $\mathbb{v} \circ e_2'$ with the same updated state δ'. This ensures that e_2 is fully evaluated before the concatenation result is computed. The [CONCAT-3] rule defines the final step of the concatenation operation. If both operands $\mathbb{v}_1$ and $\mathbb{v}_2$ are already values, and their concatenation results in the value $\mathbb{v}$, then $\mathbb{v}_1 \circ \mathbb{v}_2$ evaluates to $\mathbb{v}$ with the state δ remaining unchanged. This completes the concatenation process by combining the two values into a single result.

These rules collectively ensure that concatenation operations are evaluated in a systematic and step-by-step manner, handling both the evaluation of individual operands and the final combination of values. We also introduce a new rule for explicit substitution for concatination:

$$\text{[S-CONCAT]} \qquad (e_1 \circ e_2)\sigma \equiv (e_1\sigma) \circ (e_2\sigma)$$

Figure 4.22: Explicit substitution for concatenation.

The rule S-CONCAT states that we can perform substitution on an expression which is $e_1 \circ e_2$ by subtitution both subexpressions.

$$\text{[T-CONCAT]} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \circ e_2 : \tau_1 \circ \tau_2}$$

Figure 4.23: Type rule for concatenation.

The [T-CONCAT] rule in Figure 4.23 provides the type checking mechanism for concatenation operations in λ_{sh}. This rule states that if we have two expressions e_1 and e_2 they become concatination types and we do not forget typing information this way, since we have the type equivalence T-CONSTR presented in section 4.6. The expression $\lfloor e_1 \rfloor \circ \lfloor e_2 \rfloor$ represents the concatenation of the quoted forms of e_1 and e_2. In λ_{sh}, concatenation combines these quoted values into a single expression by the type rule T-QCON presented in section 4.6.

4.13 Expressivity

Now, with quoting established as a mechanism, we look at how it can be used. Quoting is a versatile tool which we want to show. First let us introduce a function to construct relative paths.

$$rel := \lambda x : filepath \;.\; \lfloor cwd \rfloor \circ x$$

Now, we can use the abstraction either by unquoting or composing quotes.

$$\begin{aligned} path &:= \lceil rel\ "/src" \rceil \\ cd\_src &:= \lfloor cd\ (rel\ "/src") \rfloor \\ new\_path &:= rel\ "/src" \circ "/main.c" \end{aligned}$$

Its worth noting that these example show some uses for quoting and the expressivity it provides by treating commands and macros by string interpolation. The *path* definition constructs a quote from calling the abstraction and unquotes the result (of type $filepath$), *cd_src* constructs a macro from the quote yielded by $rel\ "/src"$ passing the quoted result to cd (resulting in type $\lfloor unit \rfloor$), *new_path* is an example of simple concatination with quoted values to construct a new path (of type $\lfloor filepath \rfloor$).

4.14 Summary

This chapter has detailed the formal specification of λ_{sh}, a functional shell language designed to address the limitations of traditional shell scripting languages by incorporating a strong type system and robust operational semantics.

We began by establishing the syntactic categories and formation rules that define the structure of λ_{sh}. These foundational elements extend the traditional λ-calculus to include string interpolation, core operating system functions, and a model of the file system, which are essential for practical shell scripting. The computational model, based on the $\lambda\sigma$-calculus, emphasises explicit substitutions, offering a precise and efficient framework for computation. This model ensures that substitutions are handled explicitly, which is crucial for maintaining clarity and correctness in program evaluation. The small-step operational semantics provide a formal description of how λ_{sh} programs are executed. This formalism is key to reasoning about program behaviour and ensuring that programs execute as expected.

The type system of λ_{sh} is designed to catch type errors at compile time, enhancing the reliability of scripts, which we will prove that it does. By integrating these theoretical principles, λ_{sh} aims to offer a more reliable scripting environment than traditional shell languages, while maintaining their flexibility and ease of use. This formal specification forms the basis for implementing and reasoning about λ_{sh}, providing a structured approach to developing and maintaining shell scripts with strong type guarantees.

With this framework in place, we are now equipped to demonstrate that the type system of λ_{sh} is sound, ensuring that well-typed programs are free from type errors during execution.

Chapter 5

Soundness

5.1 Motivation

Now, we have presented the evaluation rules for λ_{sh} along with a type system to prevent type errors. A type system is said to be *sound* if well-typed programs cannot produce type errors, this is a property we want to prove for the λ_{sh} such that we have some guarantee of correctness. To prove soundness in the λ_{sh} type system, we informally define it as 2 properties which together guarantees an invariant such that a well-typed expression will never get stuck and will remain well-typed:

- **Progress:** A well-typed term is either a value or can be evaluated one step further.
- **Preservation:** If a well-typed term can be evaluated, then the result is also well-typed.

In the domain of shell programming, errors can become dangerous because we are working with the underlying operating system. In λ_{sh} have introduced a type system to prevent runtime errors however, we need to prove that progress and preservation hold. Now we begin the soundness proof.

5.2 Preservation

Lemma 1 (Substitution strengthening)**.** *If* $\Gamma \vdash e\sigma : \tau$ *then* $\Gamma \vdash e\sigma' : \tau$ *where* $\sigma' = \sigma \downarrow_{fv(e)}$.

Lemma 1 says that we can remove elements from a substitution if those variables are not part of the free variables in our expression as those should not affect the typing judgement of that expression. We are not going to prove Lemma 1, but assume that it holds.

Lemma 2 (Structural congruence preserves types)**.** *If* $e \equiv e'$*, then we we have* $\Gamma \vdash e : \tau$ *iff* $\Gamma \vdash e' : \tau$

Proof. We proceed by induction in the reduction rules for $\equiv$.

Case [S-NUM]:

$$n[\sigma] \equiv n$$

We assume $\Gamma \vdash n[\sigma] \equiv n$. We need to show $\Gamma \vdash n : \tau$. By the rule [T-CLOS] $\dfrac{\Gamma \vdash \sigma \triangleright \Gamma' \quad \Gamma' \vdash e : \tau}{\Gamma \vdash e[\sigma] : \tau}$ we have that $\Gamma \vdash n : \tau$, $\Gamma'' \vdash n : \tau$ for all Γ'' and $\tau = num$

Assume $\Gamma \vdash n : \tau$. We need to show $\Gamma \vdash n[\sigma] : \tau$. From the rule [T-CLOS] we have $\dfrac{\Gamma \vdash \sigma_{\emptyset} \triangleright \Gamma' \quad \Gamma' \vdash e : \tau}{\Gamma \vdash e[\sigma] : \tau}$, then by Lemma 1 we have $\Gamma \vdash n\sigma : \tau$ because we know $fv(n) = \emptyset$

Case [S-BOOL]: The proof for this case is similar to [S-NUM].

Case [S-STRING]: The proof for this case is similar to [S-NUM].

Case [S-UNIT]: The proof for this case is similar to [S-NUM].

Case [S-FILEPATH]: The proof for this case is similar to [S-NUM].

Case [S-CONCAT]:

$$(e_1 \circ e_2)[\sigma] \equiv (e_1[\sigma]) \circ (e_2[\sigma])$$

We assume $\Gamma \vdash (e_1 \circ e_2)[\sigma] : \texttt{string}$. We need to show $\Gamma \vdash (e_1[\sigma]) \circ (e_2[\sigma]) : \texttt{string}$.

From the typing rule `[T-CONCAT]`, we know:

$$\frac{\Gamma \vdash e_1 : \texttt{string} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1 \circ e_2 : \texttt{string}}$$

By the induction hypothesis, if $\Gamma \vdash e_1 : \texttt{string}$ and $\Gamma \vdash \sigma \triangleright \Gamma'$, then $\Gamma' \vdash e_1[\sigma] : \texttt{string}$. Similarly, if $\Gamma \vdash e_2 : \texttt{string}$ and $\Gamma \vdash \sigma \triangleright \Gamma'$, then $\Gamma' \vdash e_2[\sigma] : \texttt{string}$.

Therefore, $\Gamma \vdash (e_1[\sigma]) \circ (e_2[\sigma]) : \texttt{string}$ by the typing rule for concatenation, which states that if $\Gamma \vdash e_1 : \texttt{string}$ and $\Gamma \vdash e_2 : \texttt{string}$, then $\Gamma \vdash e_1 \circ e_2 : \texttt{string}$.

Thus, the preservation property holds for concatenation.

Case [REFL]:

$$e \equiv e$$

We use the axiom [REFL] $e \equiv e$ which holds trivially since $\Gamma \vdash e : \tau$ if and only if $\Gamma \vdash e : \tau$.

Case [SYMM]:

$$e' \equiv e$$

We use the rule [SYMM] $\dfrac{e' \equiv e}{e \equiv e'}$, by induction hypothesis we have that $\Gamma \vdash e' : \tau$ if and only if $\Gamma \vdash e : \tau$ which implies we also have $\Gamma \vdash e : \tau$ if and only if $\Gamma \vdash e' : \tau$

Case [TRANS]:

$$e \equiv e'$$

We use the rule [TRANS] $\dfrac{e_1 \equiv e \quad e_1 \equiv e'}{e \equiv e}$, by induction hypothesis we have $\Gamma \vdash e' : \tau$ if and only if $\Gamma \vdash e' : \tau$ which also implies $\Gamma \vdash e' : \tau$ if and only if $\Gamma \vdash e : \tau$

Case [ABS]:

$$(\lambda x : \tau.e)[s] \equiv \lambda y.(e[(y/x) \cdot s])$$

Assume $\Gamma \vdash \lambda x : \tau.e : \tau \to \tau'$ and $\Gamma \vdash s \triangleright \Gamma'$. By the typing rule for abstractions, we have $\Gamma[x \mapsto \tau] \vdash e : \tau'$.

When $(\lambda x : \tau.e)[s] \equiv \lambda y : \tau.(e[(y/x) \cdot s])$, we need to show $\Gamma \vdash \lambda y : \tau.(e[(y/x) \cdot s]) : \tau \to \tau'$.

By substitution, the type of $\lambda y : \tau.(e[(y/x) \cdot s])$ is $\tau \to \tau'$ since $\Gamma[y \mapsto \tau] \vdash e[(y/x) \cdot s] : \tau'$, we have:

$$\frac{\Gamma[y \mapsto \tau] \vdash e[(y/x) \cdot s] : \tau'}{\Gamma \vdash \lambda y : \tau.(e[(y/x) \cdot s]) : \tau \to \tau'}$$

Therefore, the preservation property holds for abstractions.

Case [APP]:

$$(e_1\ e_2)[\sigma] \equiv (e_1[\sigma])(e_2[\sigma])$$

We assume $\Gamma \vdash (e_1\ e_2)[\sigma] : \tau$. We need to show $\Gamma \vdash (e_1[\sigma])(e_2[\sigma]) : \tau$.

From the typing rule [T-CLOS], we have:

$$\frac{\Gamma \vdash \sigma \triangleright \Gamma' \quad \Gamma' \vdash e : \tau}{\Gamma \vdash e[\sigma] : \tau}$$

Applying this to $(e_1\ e_2)[\sigma]$:

$$\Gamma \vdash (e_1\ e_2)[\sigma] : \tau \implies \Gamma \vdash \sigma \triangleright \Gamma' \quad \Gamma' \vdash e_1\ e_2 : \tau$$

From the typing rule [T-APP], we have:

$$\frac{\Gamma' \vdash e_1 : \tau_1 \to \tau \quad \Gamma' \vdash e_2 : \tau_1}{\Gamma' \vdash e_1\ e_2 : \tau}$$

Applying the substitution σ to both e_1 and e_2:

$$\Gamma' \vdash e_1 : \tau_1 \to \tau \implies \Gamma \vdash e_1[\sigma] : \tau_1 \to \tau$$

$$\Gamma' \vdash e_2 : \tau_1 \implies \Gamma \vdash e_2[\sigma] : \tau_1$$

Therefore, by the typing rule [T-APP]:

$$\frac{\Gamma \vdash e_1[\sigma] : \tau_1 \to \tau \quad \Gamma \vdash e_2[\sigma] : \tau_1}{\Gamma \vdash (e_1[\sigma])(e_2[\sigma]) : \tau}$$

Thus, $\Gamma \vdash (e_1\ e_2)[\sigma] : \tau$ if and only if $\Gamma \vdash (e_1[\sigma])(e_2[\sigma]) : \tau$, proving the property of the lemma for the [APP] case.

The case can be extended by considering that subtyping can be applied. By induction hypothesis we have $\Gamma \vdash e_1\ e_2[\sigma] : \tau$ and $(e_1\ e_2)[\sigma] \equiv (e_1[\sigma])(e_2[\sigma])$. By the the [T-SUB] rule

$$\frac{\Gamma \vdash e : \tau_s^1 \quad \tau_s <: \tau^2}{\Gamma \vdash e : \tau}$$

We have that if (1) and (2) hold, the expression becomes well-typed by subtyping.

Case [APP-1]:

$$e_1\ e \equiv e_2\ e$$

We assume $\Gamma \vdash e_1\ e : \tau$ and we use the rule [APP-1] $\dfrac{e_1 \equiv e_2}{e_1\ e \equiv e_2\ e}$, by induction hypothesis we have $\Gamma \vdash e_2\ e : \tau$ if and only if $\Gamma \vdash e_1\ e : \tau$ which implies $\Gamma \vdash e_1\ e : \tau$ if and only if $\Gamma \vdash e_2\ e : \tau$.

Case [APP-2]:

$$e\ e_1 \equiv e\ e_2$$

We assume $\Gamma \vdash e\ e_1 : \tau$ and we use the rule [APP-2] $\dfrac{e_1 \equiv e_2}{e\ e_1 \equiv e\ e_2}$ then, by induction hyopthesis we have that $\Gamma \vdash e\ e_2 : \tau$ if and only if $\Gamma \vdash e\ e_1 : \tau$ which implies $\Gamma \vdash e\ e_1 : \tau$ if and only if $\Gamma \vdash e\ e_2 : \tau$.

Case [VAR-1]:

$$x[(e/x) \cdot s] \equiv e$$

We assume $\Gamma \vdash x : \tau$. We need to show $\Gamma \vdash e : \tau$.

Since $\Gamma(x) = \tau$ and by substitution, $x[(e/x) \cdot s] \equiv e$, e retains the type τ.

Case [VAR-2]:

$$\frac{x \neq y}{x[(e/y) \cdot s] \equiv x[s]}$$

We assume $\Gamma \vdash x : \tau$. We need to show $\Gamma \vdash x[s] : \tau$.

Since $x \neq y$, substitution does not change the variable x. Therefore, the type of $x[s]$ remains τ.

Case [VAR-3]:

$$x[\texttt{id}] \equiv x$$

We assume $\Gamma \vdash x : \tau$. We need to show $\Gamma \vdash x : \tau$.

This follows directly since applying the identity substitution does not change the variable x.

Therefore, in all cases, the preservation property holds for variables.

Case [INV]:

$$\lceil \lfloor e \rfloor \rceil \equiv e$$

We assume $\Gamma \vdash \lceil \lfloor e \rfloor \rceil : \tau$, then we have 2 subcases:

Subcases [INV]:

1. **Subcase [T-QUOTE]**

   We want to show that $\lceil \lfloor e \rfloor \rceil \equiv e$, there exists a derivation of using the rules [T-UNQUOTE] and [T-QUOTE]:

   $$\frac{\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lfloor e \rfloor : \lfloor \tau \rfloor}}{\Gamma \vdash \lceil \lfloor e \rfloor \rceil : \tau}$$

   which concludes this subcase.

2. **Subcase [T-SUB]**

   $$\frac{\Gamma \vdash \lfloor e \rfloor : \lfloor \tau \rfloor}{\Gamma \vdash \lceil \lfloor e \rfloor \rceil : \lfloor \tau \rfloor}$$

   We use the rule [T-UNQUOTE] now, by [T-SUB] we have that

   $$\frac{\Gamma \vdash e : \lfloor \tau \rfloor^1 \quad \lfloor \tau \rfloor <: string^2}{\Gamma \vdash e : string}$$

   and since (1) and (2) hold, we have $\Gamma \vdash e : string$ which concludes this subcase.

Case [IDEM]:

$$\lfloor \lfloor e \rfloor \rfloor \equiv \lfloor e \rfloor$$

We assume $\Gamma \vdash \lfloor \lfloor e \rfloor \rfloor : \tau$ then we have 2 subcases:

Subcases [IDEM]:

1. **Subcase [T-QUOTE]**

   $$\frac{\Gamma \vdash e : \tau^{(1)}}{\Gamma \vdash \lfloor e \rfloor : \lfloor \tau \rfloor}$$

   By the rule [T-QUOTE] we have, then we have that (1) holds and by induction hypothesis we have $\Gamma \vdash \lfloor \lfloor e \rfloor \rfloor : \tau_1$. $\tau = \tau_1$ and a quoted expression is a value which concludes this subcase.
2. **Subcase [T-SUB]**

   $$\frac{\Gamma \vdash e : \tau^{(*)}}{\Gamma \vdash \lfloor e \rfloor : \lfloor \tau \rfloor}$$

   Again, by the rule [T-QUOTE] we have and (*) holds and by induction hypothesis we have $\Gamma \vdash \lfloor \lfloor e \rfloor \rfloor : \tau_1$, by the rule [T-SUB] we have $\frac{\Gamma \vdash e : {\tau_s}^{(1)} \quad \tau_s <: \tau^{(2)}}{\Gamma \vdash e : \tau}$, then since (1) and (2) hold, since $\Gamma \vdash \lfloor \lfloor e \rfloor \rfloor$ and $\lfloor \tau \rfloor <: \mathit{string}$. Finally by [T-SUB] we have that $\Gamma \vdash \lfloor \lfloor e \rfloor \rfloor : \mathit{string}$ which concludes this subcase.

Case [QCON]:

$$\lfloor e_1 \rfloor \circ \lfloor e_2 \rfloor \equiv \lfloor e_1 \circ e_2 \rfloor$$

We assume $\Gamma \vdash \lfloor e_1 \rfloor \circ \lfloor e_2 \rfloor : \tau$ we again have 2 subcases:

Subcases [IDEM]:

1. **Subcase [T-CONCAT]**

   By [T-CONCAT] we have that

   $$\frac{\Gamma \vdash e_1 : \lfloor \tau_1 \rfloor \quad \Gamma \vdash e_2 : \lfloor \tau_2 \rfloor}{\Gamma \vdash \lfloor e_1 \rfloor \circ \lfloor e_2 \rfloor : \lfloor \tau_1 \rfloor \circ \lfloor \tau_2 \rfloor}$$

   Now we have $\tau = \lfloor \tau_1 \rfloor \circ \lfloor \tau_2 \rfloor$, then by [T-QUO] (type equivalence for concatenated quoted expressions) concludes this subcase by $\lfloor \tau_1 \rfloor \circ \lfloor \tau_2 \rfloor = \lfloor \tau_1 \circ \tau_2 \rfloor$.
2. **Subcase [T-SUB]**

   $$\frac{\Gamma \vdash e_1 : \lfloor \tau_1 \rfloor \quad \Gamma \vdash e_2 : \lfloor \tau_2 \rfloor}{\Gamma \vdash \lfloor e_1 \rfloor \circ \lfloor e_2 \rfloor : \lfloor \tau_1 \rfloor \circ \lfloor \tau_2 \rfloor}$$

   By [T-CONCAT] we have that Then, by [T-QUO] we have $\frac{\Gamma \vdash e_1 : {\tau_1}^{(1)}}{\Gamma \vdash \lfloor e_1 \rfloor : \lfloor \tau_1 \rfloor}$ and $\frac{\Gamma \vdash e_2 : {\tau_2}^{(2)}}{\Gamma \vdash \lfloor e_2 \rfloor : \lfloor \tau_2 \rfloor}$ since (1) and (2) hold we have that $\Gamma \vdash e_1 : \lfloor \tau_1 \rfloor$ and $\Gamma \vdash e_2 : \lfloor \tau_2 \rfloor$. Now both expressions can be subtyped by [T-SUB] by $\frac{\Gamma \vdash e : \lfloor \tau \rfloor \quad \lfloor \tau \rfloor <: \mathit{string}}{\Gamma \vdash e : \mathit{string}}$.

□

Theorem 5 (Preservation: the reduction rules preserves types)**.** *Assume that e is an expression, if* $\Gamma \vdash e : \tau$ *and* $e \to e'$*, then* $\Gamma \vdash e' : \tau$*.*

Proof. We proceed by induction over the reduction rules.

Case [CONCAT-1]:

$$\frac{\langle e_1, \delta \rangle \to \langle e_1', \delta' \rangle}{\langle e_1 \circ e_2, \delta \rangle \to \langle e_1' \circ e_2, \delta' \rangle}$$

Assume $\Gamma \vdash e_1 \circ e_2 : \texttt{string}$.

From the typing rule [T-CONCAT], we know:

$$\frac{\Gamma \vdash e_1 : \texttt{string} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1 \circ e_2 : \texttt{string}}$$

By the induction hypothesis, if $\Gamma \vdash e_1 : \texttt{string}$ and *T-CONCAT*, then $\Gamma \vdash e_1' : \texttt{string}$. Therefore, $\Gamma \vdash e_1' \circ e_2 : \texttt{string}$ since $\Gamma \vdash e_2 : \texttt{string}$ is unchanged.

Thus, the preservation property holds for [CONCAT-1].

Case [CONCAT-2]:

$$\frac{\langle e_2, \delta \rangle \to \langle e_2', \delta' \rangle}{\langle \mathbb{v} \circ e_2, \delta \rangle \to \langle \mathbb{v} \circ e_2', \delta' \rangle}$$

Assume $\Gamma \vdash \mathbb{v} \circ e_2 : \texttt{string}$.

From the typing rule [T-CONCAT], we know:

$$\frac{\Gamma \vdash \mathbb{v} : \texttt{string} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash \mathbb{v} \circ e_2 : \texttt{string}}$$

By the induction hypothesis, if $\Gamma \vdash e_2 : \texttt{string}$ and $\langle e_2, \delta \rangle \to \langle e_2', \delta' \rangle$, then $\Gamma \vdash e_2' : \texttt{string}$. Therefore, $\Gamma \vdash \mathbb{v} \circ e_2' : \texttt{string}$ since $\Gamma \vdash \mathbb{v} : \texttt{string}$ is unchanged.

Thus, the preservation property holds for [CONCAT-2].

Case [CONCAT-3]:

$$\frac{}{\langle \mathbb{v}_1 \circ \mathbb{v}_2, \delta \rangle \to \langle \mathbb{v}, \delta \rangle} \text{ if } \mathbb{v} = \mathbb{v}_1 \circ \mathbb{v}_2$$

Assume $\Gamma \vdash \mathbb{v}_1 \circ \mathbb{v}_2 : \texttt{string}$.

From the typing rule [T-CONCAT], we know:

$$\frac{\Gamma \vdash \mathbb{v}_1 : \texttt{string} \quad \Gamma \vdash \mathbb{v}_2 : \texttt{string}}{\Gamma \vdash \mathbb{v}_1 \circ \mathbb{v}_2 : \texttt{string}}$$

Since $\mathbb{v} = \mathbb{v}_1 \circ \mathbb{v}_2$, $\mathbb{v}$ is of type `string`.

Thus, the preservation property holds for [CONCAT-3].

Case [STRUCT]: By Lemma 2, we have shown that structural congruence preserves types.

Case [CWD]:

$$\frac{}{\langle \texttt{cwd}, \delta \rangle \to \langle \mathbb{p}, \delta \rangle} \text{ where } \delta = (\mathbb{p}, \texttt{fs})$$

Assuming $\Gamma \vdash \texttt{cwd} : \texttt{filepath}$. We need to show $\Gamma \vdash \mathbb{p} : \texttt{filepath}$. Since $\delta = (\texttt{cwd}, \texttt{fs})$ and $\mathbb{p} = \texttt{cwd}$ and $\mathbb{p}$ is of type `filepath`, $\Gamma \vdash \mathbb{p} : \texttt{filepath}$ holds.

Thus, the preservation property holds for [CWD].

Case [MKDIR-1]:

$$\frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{mkdir } e, \delta \rangle \to \langle \texttt{mkdir } e', \delta' \rangle}$$

We assume $\Gamma \vdash \texttt{mkdir}\ e : \texttt{unit}$. We need to show $\Gamma \vdash \texttt{mkdir}\ e' : \texttt{unit}$. From the typing rule, we know:

$$\frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{mkdir}\ e : \texttt{unit}}$$

By the induction hypothesis, if $\Gamma \vdash e : \texttt{filepath}$ and $\langle e, \delta \rangle \to \langle e', \delta' \rangle$, then $\Gamma \vdash e' : \texttt{filepath}$. Therefore, $\Gamma \vdash \texttt{mkdir}\ e' : \texttt{unit}$.

Thus, the preservation property holds for [MKDIR-1].

Case [MKDIR-2]:

$$\frac{}{\langle \texttt{mkdir}\ \mathbb{p}, (\mathbb{p}', \texttt{fs}) \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ if } \delta' = (\mathbb{p}', \texttt{fs} \cup \mathbb{p})$$

We assume $\Gamma \vdash \texttt{mkdir}\ e : \texttt{unit}$. We need to show $\Gamma \vdash \texttt{unit} : \texttt{unit}$. Since $\texttt{mkdir}\ \mathbb{p}$ results in the unit type, $\Gamma \vdash \texttt{unit} : \texttt{unit}$ holds trivially. Thus, the preservation property holds for [MKDIR-2].

Case [RM-1]:

$$\frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{rm}\ e, \delta \rangle \to \langle \texttt{rm}\ e', \delta' \rangle}$$

We assume $\Gamma \vdash \texttt{rm}\ e : \texttt{unit}$. We need to show $\Gamma \vdash \texttt{rm}\ e' : \texttt{unit}$. From the typing rule, we know:

$$\frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{rm}\ e : \texttt{unit}}$$

By the induction hypothesis, if $\Gamma \vdash e : \texttt{filepath}$ and $\langle e, \delta \rangle \to \langle e', \delta' \rangle$, then $\Gamma \vdash e' : \texttt{filepath}$. Therefore, $\Gamma \vdash \texttt{rm}\ e' : \texttt{unit}$. Thus, the preservation property holds for [RM-1].

Case [RM-2]:

$$\frac{}{\langle \texttt{rm}\ \mathbb{p}, (\mathbb{p}', \texttt{fs}) \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ if } \delta' = (\mathbb{p}', \texttt{fs} \setminus \mathbb{p})$$

Assuming $\Gamma \vdash \texttt{rm}\ e : \texttt{unit}$, we need to show $\Gamma \vdash \texttt{unit} : \texttt{unit}$. Since $\texttt{rm}\ \mathbb{p}$ results in the unit type, $\Gamma \vdash \texttt{unit} : \texttt{unit}$ holds trivially. Thus, the preservation property holds for [RM-2].

Case [TOUCH-1]:

$$\frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{touch}\ e, \delta \rangle \to \langle \texttt{touch}\ e', \delta' \rangle}$$

We assume $\Gamma \vdash \texttt{touch}\ e : \texttt{unit}$. We need to show $\Gamma \vdash \texttt{touch}\ e' : \texttt{unit}$. From the typing rule, we know:

$$\frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{touch}\ e : \texttt{unit}}$$

By the induction hypothesis, if $\Gamma \vdash e : \texttt{filepath}$ and $\langle e, \delta \rangle \to \langle e', \delta \rangle$, then $\Gamma \vdash e' : \texttt{filepath}$. Therefore, $\Gamma \vdash \texttt{touch}\ e' : \texttt{unit}$.

Thus, the preservation property holds for [TOUCH-1].

Case [TOUCH-2]:

$$\frac{}{\langle \texttt{touch}\ \mathbb{p}, (\mathbb{p}', \texttt{fs}) \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ if } \delta' = (\mathbb{p}', \texttt{fs} \cup \mathbb{p})$$

We assume $\Gamma \vdash \texttt{touch}\ e : \texttt{unit}$. We need to show $\Gamma \vdash \texttt{unit} : \texttt{unit}$. Since $\texttt{touch}\ \mathbb{p}$ results in the unit type, $\Gamma \vdash \texttt{unit} : \texttt{unit}$ holds trivially.

Thus, the preservation property holds for [TOUCH-2].

Case [CD-1]:

$$\frac{\langle e, \delta\rangle \to \langle e', \delta'\rangle}{\langle \texttt{cd } e, \delta\rangle \to \langle \texttt{cd e'}, \delta\rangle}$$

Assuming $\Gamma \vdash \texttt{cd } e : \texttt{unit}$, we need to show $\Gamma \vdash \texttt{cd } e' : \texttt{unit}$.

From the typing rule, we know:

$$\frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{cd } e : \texttt{unit}}$$

By the induction hypothesis, if $\Gamma \vdash e : \texttt{filepath}$ and $\langle e, \delta \to \langle e', \delta'\rangle$, then $\Gamma \vdash e' : \texttt{filepath}$. Therefore, $\Gamma \vdash \texttt{cd } e' : \texttt{unit}$.

Thus, the preservation property holds for [CD-1].

Case [CD-2]:

$$\frac{}{\langle \texttt{cd } \mathbb{p}, \delta\rangle \to \langle \mathbb{u}, \delta'\rangle} \text{ where } \delta' = (\mathbb{p}, \texttt{fs})$$

We assume $\Gamma \vdash \texttt{cd } e : \texttt{unit}$. We need to show $\Gamma \vdash \texttt{unit} : \texttt{unit}$. Since $\texttt{cd } \mathbb{p}$ results in the unit type, $\Gamma \vdash \texttt{unit} : \texttt{unit}$ holds trivially. Thus, the preservation property holds for [CD-2].

□

Now we have shown that the progress property holds for the $\lambda_s h$ type system. Now we also need to show progress for λ_{sh} in order to complete the soundness proof.

5.3 Progress

Theorem 6 (Progress: an expression is either a value or can be evaluated further)**.** *Assume that e is and expression. If $\Gamma \vdash e : \tau$, then e is either a value or there exists an e' such that $e \to e'$.*

Proof. We proceed by induction on the derivation of the typing judgement $\Gamma \vdash e : T$.

Case [T-NUM]:

$$\frac{}{\Gamma \vdash n : \texttt{num}}$$

A numeral literal is already a value.

Case [T-BOOL]:

$$\frac{}{\Gamma \vdash b : \texttt{bool}}$$

A boolean literal is already a value.

Case [T-STRING]:

$$\frac{}{\Gamma \vdash s : \texttt{string}}$$

A string literal is already a value.

Case [T-FILEPATH]:

$$\frac{}{\Gamma \vdash p : \texttt{filepath}}$$

A filepath literal is already a value.

Case [T-UNIT]:

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{unit}}$$

The unit value `unit` is already a value.

Case [T-VAR]:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Variables are assumed to be mapped to values in the environment, so they can take a step to their corresponding value.

Case [T-ABS]:

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.e) : \tau \to \tau'}$$

A lambda abstraction $(\lambda x : \tau.e)$ is already a value.

Case [T-APP]:

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \tau'}$$

If e_1 is not a value, then by the induction hypothesis, there exists e_1' such that $\langle e_1, \delta\rangle \to \langle e_1', \delta'\rangle$. Hence, $\langle e_1\, e_2, \delta\rangle \to \langle e_1'\, e_2, \delta\rangle$. Similarly, if e_2 is not a value, then $\langle e_2, \delta\rangle \to \langle e_2', \delta'\rangle$ and $\langle e_1\, e_2, \delta\rangle \to \langle e_1\, e_2', \delta'\rangle$. If both e_1 and e_2 are values, then by the [BETA] rule, $(\langle \lambda x.e_1)e_2, \delta\rangle \to \langle e_1[e_2/x], \delta\rangle$.

Case [T-CLOS]:

$$\frac{\Gamma \vdash \sigma \triangleright \Gamma' \quad \Gamma' \vdash e : \tau}{\Gamma \vdash e[\sigma] : \tau}$$

If e is not a value, then by the induction hypothesis, there exists e' such that $\langle e, \delta\rangle \to \langle e', \delta'\rangle$. Therefore, $\langle e[\sigma], \delta\rangle \to \langle e'[\sigma], \delta\rangle$.

Case [T-ID]:

$$\frac{}{\Gamma \vdash \texttt{id} \triangleright \Gamma}$$

The identity substitution `id` does not change the expression to which it is applied. Therefore, `id` does not transition and is already in a terminal configuration.

Case [T-CONS]:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \sigma \triangleright \Gamma'}{\Gamma \vdash (e/x) \cdot \sigma \triangleright \Gamma'[x \mapsto \tau]}$$

This rule is about typing substitutions. It indicates that if we have a substitution of the form $(e/x)\cdot\sigma$, then we need to ensure e has the type τ and σ is a valid substitution. If $\Gamma' \vdash e : \tau$, then e is either a value or there exists e' such that $\langle e, \delta\rangle \to \langle e', \delta'\rangle$. Then we have either of two options:

1. e is a value. In this case, $e[\sigma]$ remains a value.
2. $\langle e, \delta\rangle \to \langle e', \delta'\rangle$. By the induction hypothesis, since e can take a step to e', the substitution application respects this transition:

   $$\langle e[\sigma], \delta\rangle \to \langle e'[\sigma], \delta\rangle$$

   Thus, $e[\sigma]$ either progresses to $e'[\sigma]$ or remains a value.

For the progress case involving substitution [T-SUB], we confirm that the application of a substitution to an expression either results in a value or in an expression that can take a transition step. Substitutions do not take transitions themselves but ensure the resultant expression adheres to the transition rules.

Case [T-APP-FILEPATH]:

$$\frac{\Gamma \vdash e_1 : \texttt{filepath} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1\ e_2 : \texttt{unit}}$$

If e_1 is not a value, then by the induction hypothesis, there exists e_1' such that $\langle e_1, \delta \rangle \to \langle e_1', \delta' \rangle$. Hence, $\langle e_1\, e_2, \delta \rangle \to \langle e_1'\, e_2, \delta' \rangle$. Similarly, if e_2 is not a value, then $\langle e_2, \delta \rangle \to \langle e_2', \delta' \rangle$ and $\langle e_1\, e_2, \delta \rangle \to \langle e_1\, e_2', \delta' \rangle$.

Case [T-APP-STRING]:

$$\frac{\Gamma \vdash e_1 : \texttt{string} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1\, e_2 : \texttt{unit}}$$

If e_1 is not a value, then by the induction hypothesis, there exists e_1' such that $\langle e_1, \delta \rangle \to \langle e_1', \delta' \rangle$. Hence, $\langle e_1\, e_2, \delta \rangle \to \langle e_1'\, e_2, \delta' \rangle$. Similarly, if e_2 is not a value, then $\langle e_2, \delta \rangle \to \langle e_2', \delta' \rangle$ and $\langle e_1\, e_2, \delta \rangle \to \langle e_1\, e_2', \delta' \rangle$.

Case [T-CONCAT]:

$$\frac{\Gamma \vdash e_1 : \texttt{string} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1 \circ e_2 : \texttt{string}}$$

If e_1 is not a value, then by the induction hypothesis, there exists e_1' such that $\langle e_1, \delta \rangle \to \langle e_1', \delta' \rangle$. Hence, $\langle e_1 \circ e_2, \delta \rangle \to \langle e_1' \circ e_2, \delta' \rangle$ from rule [CONCAT-1]. Similarly, if e_2 is not a value, then $\langle e_2, \delta \rangle \to \langle e_2', \delta' \rangle$ and $\langle e_1 \circ e_2, \delta \rangle \to \langle e_1 \circ e_2', \delta' \rangle$ from rule [CONCAT-2]. If both e_1 and e_2 are values, then by the [CONCAT-3] rule, $\langle e_1 \circ e_2, \delta \rangle \to \langle v, \delta' \rangle$, where v is a string value.

Case [T-QUOTE]:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lfloor e \rfloor : \lfloor \tau \rfloor}$$

Quotes ($\lfloor e \rfloor$) are considered values in the language. Therefore, when we encounter an expression of the form $\lfloor e \rfloor$, it is inherently a value, regardless of whether e itself is a value or can take a step. This is because the act of quoting e encapsulates it as a value within the quote.

In other words, the expression $\lfloor e \rfloor$ does not need to transition further because it is treated as an atomic value in its quoted form. As a result, there is no need for an evaluation rule to further transition $\lfloor e \rfloor$. This directly ensures the progress property, as $\lfloor e \rfloor$ will never be in a state where it needs to take a step to transition to another expression.

Case [T-UNQUOTE]:

$$\frac{\Gamma \vdash e : \lfloor \tau \rfloor}{\Gamma \vdash \lceil e \rceil : \tau}$$

If e is a value, specifically of the form $\lfloor v \rfloor$, then by the rule [INV]:

$$\langle \lceil \lfloor e \rfloor \rceil, \delta \rangle \to \langle e, \delta \rangle$$

Hence, $\lceil e \rceil$ progresses directly to e.

If e is not a value, then by the induction hypothesis, there exists e' such that $\langle e, \delta \rangle \to \langle e', \delta' \rangle$. Therefore, $\langle \lceil e \rceil, \delta \rangle \to \langle \lceil e' \rceil, \delta' \rangle$.

Case [T-CWD]:

$$\frac{}{\Gamma \vdash \texttt{cwd} : \texttt{filepath}}$$

The current working directory `cwd` is already a value of type `filepath`.

Case [T-CD]:

$$\frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{cd}\ e : \texttt{unit}}$$

If e is not a value, then by the induction hypothesis, there exists e' such that $\langle e, \delta \rangle \to \langle e', \delta' \rangle$. Hence, $\langle \texttt{cd}\ e, \delta \rangle \to \langle \texttt{cd}\ e', \delta' \rangle$ by [CD-1]. If e in `cd` e is a value, then there exists the transition such that $\langle \texttt{cd}\ e, \delta \rangle \to \langle \mathbb{u}, \delta' \rangle$ by [CD-2].

The proof is roughly the same for the [T-TOUCH], [T-MKDIR], and [T-RM] cases.

That is the end of the progress proof. □

5.4 Summary

We have now established both the progress and preservation properties for the λ_{sh} language, demonstrating that well-typed expressions are either values or can take a step according to the evaluation rules, and that types are preserved during evaluation (we will never become stuck in a non-terminal configuration).

5.4.1 Progress

For each case of the type rules, we have shown that a well-typed expression is either a value or can transition to another expression:

- For base types like numerals, booleans, strings, filepaths, and units, we have shown that these are values.
- For complex constructs such as lambda abstractions, applications, substitutions, concatenations, quotes, and unquotes, we have demonstrated that they either are values or can make progress according to the semantics.

5.4.2 Preservation

For each transition step in the evaluation, we have shown that the type of the expression is preserved:

- Application of functions and substitutions maintains the type consistency of the expressions.
- Structural congruence and rewriting rules ensure that transitions between equivalent forms preserve types.
- Special constructs like quoting and unquoting were shown to correctly handle type transformations without violating the type rules.

5.4.3 Final assertion of soundness

With both progress and preservation established, we conclude that the λ_{sh} languages type system is sound. This means that if a term is well-typed, it will always be well-typed and never be stuck. Therefore, the soundness of the type system guarantees the reliability of λ_{sh}'s behaviour as specified by its typing and evaluation rules. This ensures that well-typed programs in λ_{sh} will execute correctly without type-related runtime errors, providing a solid foundation for the language's correctness and robustness.

Chapter 6

Conclusion, discussion, and future work

6.1 Conclusion

In this thesis, we have introduced λ_{sh}, an approach to implementing a strong type system in a shell programming language where strings play a central role. Traditional shell scripting languages, such as Bash and PowerShell, are powerful tools for automation but often suffer from weak typing systems, leading to runtime errors, maintenance difficulties, and security vulnerabilities. Our work on λ_{sh} addresses these issues by leveraging the principles of λ-calculus and formal type theory to create a robust, statically-typed formal specification for a shell language.

The key contributions of this thesis include:

1. **Design and semantics of λ_{sh}:** We have defined the abstract syntax, operational semantics, and type system of λ_{sh}, ensuring that the language supports predictable and consistent handling of various features from shell scripting and string handling.
2. **Formal analysis:** Through formal proofs, we have demonstrated the soundness of the type system, ensuring that well-typed programs do not encounter type-related runtime errors.
3. **Foundation for future extensions:** We have focused on establishing a solid foundation for λ_{sh}, which can be extended with more advanced features such as records, namespacing, and concurrency constructs.

In conclusion, the work presented in this thesis addresses critical weaknesses in traditional shell scripting languages and sets the stage for the development of more reliable and secure shell programming tools. While the formal specification still requires some work in order to be a fully fledged language specification for a modern programming language, we are satisfied with ground work presented in this thesis, which can be built and improved upon.

6.2 Discussion

While the development of λ_{sh} has established a foundation for a strongly-typed shell programming language, there are areas for further enhancement. Specifically, the language could be more feature-rich, and the modelling of the shell environment could be more detailed.

6.2.1 More features

Although the primary focus of this project was on fundamental aspects, λ_{sh} could benefit from additional features common in modern programming languages, such that the language specification could feel more

complete in the sense that it documents the features you would expect in a modern programming language:

- **Data structures**: Incorporating records, tuples, and lists would allow for more structured data handling and manipulation through abstraction.
- **Namespacing and modules**: Implementing namespacing and modularity would prevent naming conflicts and support better code organisation and reuse. Something that is difficult to do well in traditional shells.
- **Error handling constructs**: Advanced error handling constructs are crucial for creating robust and fault-tolerant scripts. This was originally planned to be a feature, but getting to having it proved more cumbersome. Here our inspiration was to add a better error handling system to shell languages, through exceptions or some other error mechanism, since traditional shell languages have awful error handling mechanisms, since they involve dealing with return codes explicitly.
- **Concurrency and parallelism**: Adding constructs for concurrency and parallelism would model the inherently parallel nature of shell languages. Here, it would be interesting to how that behaviour modelled in a formal semantics, such that it can be reasoned about. Also, this seems like an important thing to model since one of shells main functions is to deal with processes and jobs.

The idea is that the language needs to capture the entirety of a language. Right now, λ_{sh} may be a bit more like a single part of a larger whole, as the language ends up being an expression based concept, but obviously needs more constructs in order to be complete. We would argue that this is alright, since the purpose of the language is to showcase our take on a computational model with a type system for shell languages, where strings play a large role, and to demonstrate the quoting system.

6.2.2 Detailed shell environment modelling

The current version of λ_{sh} captures basic shell operations but lacks comprehensive environment modelling. For improved practicality, the following should be considered:

- **Filesystem Representation**: A more detailed representation, including permissions, links, and metadata, would allow for more complex file operations and interactions, such that our model better reflects the reality of the file system.
- **Process management**: Modelling of job control, background processes, and signal handling would provide a more accurate representation of Unix-like shells.
- **Environmental variables and configuration**: A thorough implementation of environmental variables and configurations would mimic real-world shell behaviour, improving the language's practicality.

By expanding language features and providing a more detailed model of the shell environment, λ_{sh} could become a more powerful and practical tool, capable of handling a broader range of scripting tasks more effectively.

6.2.3 Better leveraging of the small-step semantics

As the language stands, we don't gain significant advantages from using small-step semantics instead of big-step semantics. This is primarily because we haven't yet modelled any concurrent behaviour, which was part of the original plan for λ_{sh} to capture the parallel nature of shell languages. Given the current

features, a big-step semantics might have been simpler to implement, as the reduction rules would have been simpler to formulate, and notable, there would have been a lot less reduction rules. Although we may not fully benefit from small-step semantics by the end of this project, having this foundation will make it easier to extend λ_{sh} with concurrent features in the future.

6.3 Future work

In this paper, we have presented λ_{sh}, an approach to implementing a strong type system in a shell programming language where strings play a large role. While this work lays a solid foundation, there are several directions for future research and development to further enhance λ_{sh} and its applicability.

Expand on the semantics to model the shell environment more completely The current semantics of λ_{sh} provides a robust framework for handling strings and basic type safety. However, the shell environment encompasses a wide range of functionalities and interactions that are not yet fully modelled. Future work should aim to:

1. **Incorporate a more comprehensive representation of file systems:** This includes modelling file permissions, directories, symbolic links, and various file operations more accurately.
2. **Extend process management semantics:** Model the creation, management, and termination of processes, including background and foreground processes, job control, and inter-process communication.
3. **Improve handling of input/output streams:** Enhance the semantics to better model standard input, output, and error streams, including piping, redirection, and the use of subshells, since this is something that plays a large role in external commands.

By expanding the semantics to cover these aspects, λ_{sh} can more accurately reflect the real-world complexities of shell environments, making it a more practical and powerful tool for users.

Expand on language features The primary focus of this paper has been to establish the fundamental aspects of λ_{sh}. However, there are numerous advanced language features that could enhance the usability and expressiveness of λ_{sh}:

1. **Records and Data Structures:** Introduce more complex data structures such as records, tuples, and lists to allow more structured data handling and manipulation.
2. **Namespacing and Modules:** Implement namespacing mechanisms to prevent naming conflicts and support modular programming, making scripts more organized and maintainable.
3. **Error Handling Constructs:** Develop advanced error handling constructs to manage exceptions and errors more gracefully within scripts. This was an original goal for λ_{sh} that we didn't get to.
4. **Concurrency and Parallelism:** Add constructs for concurrent and parallel execution.

These features will not only expand the language's capabilities but also make it more realistic as a formal specification of modern shell programming language.

Implementation of a shell language following the λ_{sh} specification To validate the theoretical foundations and practical applicability of λ_{sh}, an actual implementation of the language is essential to validate the specification in reality. Future work should involve:

1. **Developing a compiler or interpreter:** Create a compiler or interpreter that adheres to the λ_{sh} specification, ensuring that the strong type system and other language features are correctly implemented.

2. **Creating a standard library:** Develop a comprehensive standard library to support common scripting tasks, such as file manipulation, text processing, and system operations.

3. **Real-world testing and feedback:** Communicate with potential users to test the implementation in real-world scenarios, gather feedback, and improve the specification.

By undertaking these steps, the λ_{sh} language can transition from a theoretical framework to a practical tool that offers real benefits to its potential users.

References

[1] M. Abadi et al. "Explicit substitutions". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. San Francisco, California, USA: Association for Computing Machinery, 1989, pp. 31–46. ISBN: 0897913434. DOI: 10.1145/96709.96712. URL: https://doi.org/10.1145/96709.96712.

[2] Christoffer Lind Andersen and Nikolai Aaen Bonderup. *Requirements for a functional shell programming language*. Technical report. Aalborg University, Jan. 2024. URL: mailto:clan19@student.aau.dk,mailto:nbonde19@student.aau.dk.

[3] *Bash Reference Manual*. 5.2. GNU. Sept. 2022.

[4] Haskell B. Curry, Robert Feys, and William Craig. "Combinatory Logic, Volume I". In: *Philosophical Review* 68.4 (1959), pp. 548–550. DOI: 10.2307/2182503.

[5] IBM Developer. Accessed: 2024. URL: https://developer.ibm.com/tutorials/l-linux-shells/.

[6] Yiwen Dong et al. "Bash in the Wild: Language Usage, Code Smells, and Bugs". In: 32.1 (Feb. 2023). ISSN: 1049-331X. DOI: 10.1145/3517193. URL: https://doi.org/10.1145/3517193.

[7] E. Engeler. "H. P. Barendregt. The lambda calculus. Its syntax and semantics. Studies in logic and foundations of mathematics, vol. 103. North-Holland Publishing Company, Amsterdam, New York, and Oxford, 1981, xiv + 615 pp." In: *Journal of Symbolic Logic* 49.1 (1984), pp. 301–303. DOI: 10.2307/2274112.

[8] Hans Huttel. *Transitions and trees*. Cambridge, England: Cambridge University Press, Oct. 2012.

[9] Mark Lutz. *Learning Python*. en. 5th ed. Sebastopol, CA: O'Reilly Media, June 2013.

[10] OpenAI DALL-E. *A strong metal seashell*. Image generated by OpenAI's DALL-E. Generated: 11-01-2024. 2024.

[11] Benjamin C Pierce. *Types and Programming Languages*. en. The MIT Press. London, England: MIT Press, Jan. 2002.

[12] Andrew K. Wright and Matthias Felleisen. "A Syntactic Approach to Type Soundness". In: *Inf. Comput.* 115 (1994), pp. 38–94. URL: https://api.semanticscholar.org/CorpusID:31415217.

Appendix A

Syntactic categories and formation rules

Syntactic categories

$n \in \mathbf{N}$	(Numerals)
$b \in \mathbf{B}$	(Booleans)
$s \in \mathbf{S}$	(String)
$p \in \mathbf{P}$	(Filepath)
$o \in \mathbf{O}$	(OS-level Constants)
$x \in \mathbf{V}$	(Variables)
$e \in \mathbf{E}$	(Expressions)
$\tau \in \mathbf{T}$	(Types)
$\sigma \in \mathbf{M}$	(Substitutions)

Formation rules

$$e ::= n \mid b \mid s \mid p \mid o \mid x \mid \lambda x : \tau . e \mid e[\sigma] \mid e_1 \circ e_2 \mid \lfloor e \rfloor \mid \lceil e \rceil$$

$$\sigma ::= \texttt{id} \mid (e/x) \cdot \sigma$$

Appendix B

Semantics

Transition system

The transition system for evaluating λ_{sh} is defined as a triple $(C, \rightarrow, T)$, where C is the configuration space, $\rightarrow$ is the transition relation, and T is the set of terminal configurations.

$$C = (\mathbf{E} \times \Delta) \cup (\mathbb{V} \times \Delta)$$
$$T = \mathbb{V} \times \Delta$$

Structural congruence

Rewriting rules

$$\text{[STRUCT]} \qquad \frac{e \equiv e_1 \quad \langle e_1, \delta \rangle \rightarrow \langle e_1', \delta' \rangle \quad e' \equiv e_1'}{\langle e, \delta \rangle \rightarrow \langle e', \delta' \rangle}$$

Structural equivalences

[BETA] $$(\lambda x.e_1)e_2 \equiv e_1[(e_2/x)]$$

[VAR-1] $$x[(e/x) \cdot s] \equiv e$$

[VAR-2] $$x[(e/y) \cdot s] \equiv x[s]$$

[VAR-3] $$x[\mathtt{id}] \equiv x$$

[APP] $$(e_1\ e_2)[\sigma] \equiv (e_1[\sigma])(e_2[\sigma])$$

[ABS] $$(\lambda x.e)[s] \equiv \lambda y.(e[(y/x) \cdot s])$$

[APP-1] $$\frac{e_1\ e \equiv e_2\ e}{e_1 \equiv e_2}$$

[APP-2] $$\frac{e\ e_1 \equiv e\ e_2}{e_1 \equiv e_2}$$

[SYMM] $$\frac{e' \equiv e}{e \equiv e'}$$

[TRANS] $$\frac{e \equiv e_1 \quad e_1 \equiv e'}{e \equiv e'}$$

[APP-LEFT] $$\frac{\langle e_1, \delta\rangle \to \langle e_1', \delta'\rangle}{\langle e_1\, e_2, \delta\rangle \to \langle e_1'\, e_2, \delta'\rangle}$$

[APP-RIGHT] $$\frac{\langle e_2, \delta\rangle \to \langle e_2', \delta'\rangle}{\langle \mathbb{v}_1\, e_2, \delta\rangle \to \langle \mathbb{v}_1\, e_2', \delta'\rangle}$$

[APP-FILEPATH] $$\frac{}{\langle \mathbb{p}\ \mathbb{s}, \delta\rangle \to \langle \mathbb{u}, \delta\rangle}$$

[APP-STRING] $$\frac{}{\langle \mathbb{s}\ \mathbb{s}, \delta\rangle \to \langle \mathbb{u}, \delta\rangle}$$

[REFL] $$e \equiv e$$

[INV] $$\lceil \lfloor e \rfloor \rceil \equiv e$$

[IDEM] $$\lfloor \lfloor e \rfloor \rfloor \equiv \lfloor e \rfloor$$

[QCON] $$\lfloor e_1 \rfloor \circ \lfloor e_2 \rfloor \equiv \lfloor e_1 \circ e_2 \rfloor$$

[S-NUM] $$n[\sigma] \equiv n$$

[S-BOOL] $$b[\sigma] \equiv b$$

[S-STRING] $$s[\sigma] \equiv s$$

[S-FILEPATH] $$p[\sigma] \equiv p$$

[S-UNIT] $$\texttt{unit}[\sigma] \equiv \texttt{unit}$$

[S-CONCAT] $$(e_1 \circ e_2)\sigma \equiv (e_1\sigma) \circ (e_2\sigma)$$

Reduction rules

[NUM] $$\frac{}{\langle n, \delta\rangle \to \langle \mathbb{n}, \delta\rangle} \text{ if } \gamma(n) = \mathbb{n}$$

[BOOL] $$\frac{}{\langle b, \delta\rangle \to \langle \mathbb{b}, \delta\rangle} \text{ if } \gamma(b) = \mathbb{b}$$

[STRING] $$\frac{}{\langle s, \delta\rangle \to \langle \mathbb{s}, \delta\rangle} \text{ if } \gamma(s) = \mathbb{s}$$

[FILEPATH] $$\frac{}{\langle p, \delta\rangle \to \langle \mathbb{p}, \delta\rangle} \text{ if } \gamma(p) = \mathbb{p}$$

[UNIT] $$\frac{}{\langle unit, \delta\rangle \to \langle \mathbb{u}, \delta\rangle}$$

[CONCAT-1] $$\frac{\langle e_1, \delta\rangle \to \langle e_1', \delta'\rangle}{\langle e_1 \circ e_2, \delta\rangle \to \langle e_1' \circ e_2, \delta'\rangle}$$

[CONCAT-2] $$\frac{\langle e_2, \delta\rangle \to \langle e_2', \delta'\rangle}{\langle \mathbb{v} \circ e_2, \delta\rangle \to \langle \mathbb{v} \circ e_2', \delta'\rangle}$$

[CONCAT-3] $$\frac{}{\langle \mathbb{v}_1 \circ \mathbb{v}_2, \delta\rangle \to \langle \mathbb{v}, \delta\rangle} \text{ if } \mathbb{v} = \mathbb{v}_1 \circ \mathbb{v}_2$$

[CWD] $$\frac{}{\langle \texttt{cwd}, \delta \rangle \to \langle \mathbb{p}, \delta \rangle} \text{ where } \delta = (\mathbb{p}, \texttt{fs})$$

[CD-1] $$\frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{cd } e, \delta \rangle \to \langle \texttt{cd e'}, \delta \rangle}$$

[CD-2] $$\frac{}{\langle \texttt{cd } \mathbb{p}, \delta \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ where } \delta' = (\mathbb{p}, \texttt{fs})$$

[TOUCH-1] $$\frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{touch } e, \delta \rangle \to \langle \texttt{touch } e', \delta' \rangle}$$

[TOUCH-2] $$\frac{}{\langle \texttt{touch } \mathbb{p}, (\mathbb{p}', \texttt{fs}) \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ if } \delta' = (\mathbb{p}', \texttt{fs}[\mathbb{p} \mapsto fval])$$

[MKDIR-1] $$\frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{mkdir } e, \delta \rangle \to \langle \texttt{mkdir } e', \delta' \rangle}$$

[MKDIR-2] $$\frac{}{\langle \texttt{mkdir } \mathbb{p}, (\mathbb{p}', \texttt{fs}) \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ if } \delta' = (\mathbb{p}', \texttt{fs}[\mathbb{p} \mapsto fval])$$

[RM-1] $$\frac{\langle e, \delta \rangle \to \langle e', \delta' \rangle}{\langle \texttt{mkdir } e, \delta \rangle \to \langle \texttt{mkdir } e', \delta' \rangle}$$

[RM-2] $$\frac{}{\langle \texttt{rm } \mathbb{p}, (\mathbb{p}', \texttt{fs}) \rangle \to \langle \mathbb{u}, \delta' \rangle} \text{ if } \delta' = (\mathbb{p}', \texttt{fs}[\mathbb{p} \mapsto \texttt{undef}])$$

Appendix C

Type system

$$\tau_1, \tau_2 ::= \texttt{bool} \mid \texttt{string} \mid \texttt{num} \mid \texttt{filepath} \mid \texttt{unit} \mid \tau_1 \to \tau_2 \mid \lfloor \tau_1 \rfloor \mid \tau_1 \circ \tau_2$$

[T-NUM] $$\frac{}{\Gamma \vdash n : \texttt{num}}$$

[T-BOOL] $$\frac{}{\Gamma \vdash b : \texttt{bool}}$$

[T-STRING] $$\frac{}{\Gamma \vdash s : \texttt{string}}$$

[T-FILEPATH] $$\frac{}{\Gamma \vdash p : \texttt{filepath}}$$

[T-UNIT] $$\frac{}{\Gamma \vdash u : \texttt{unit}}$$

[T-VAR] $$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

[T-ABS] $$\frac{\Gamma[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau . e) : \tau \to \tau'}$$

[T-APP] $$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \tau'}$$

[T-CLOS] $$\frac{\Gamma \vdash \sigma \triangleright \Gamma' \quad \Gamma' \vdash e : \tau}{\Gamma \vdash e[\sigma] : \tau}$$

[T-ID] $$\frac{}{\Gamma \vdash \texttt{id} \triangleright \Gamma}$$

[T-CONS] $$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \sigma \triangleright \Gamma'}{\Gamma \vdash (e/x) \cdot \sigma \triangleright \Gamma'[x \mapsto \tau]}$$

[T-APP-FILEPATH] $$\frac{\Gamma \vdash e_1 : \texttt{filepath} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1\ e_2 : \texttt{unit}}$$

[T-APP-STRING] $$\frac{\Gamma \vdash e_1 : \texttt{string} \quad \Gamma \vdash e_2 : \texttt{string}}{\Gamma \vdash e_1\, e_2 : \texttt{unit}}$$

$$\text{[T-CONCAT]} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \circ e_2 : \tau_1 \circ \tau_2}$$

$$\text{[T-QUOTE]} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \lfloor e \rfloor : \lfloor \tau \rfloor}$$

$$\text{[T-UNQUOTE]} \quad \frac{\Gamma \vdash e : \lfloor \tau \rfloor}{\Gamma \vdash \lceil e \rceil : \tau}$$

$$\text{[T-CWD]} \quad \frac{}{\Gamma \vdash \texttt{cwd} : \texttt{filepath}}$$

$$\text{[T-CD]} \quad \frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{cd}\ e : \texttt{unit}}$$

$$\text{[T-TOUCH]} \quad \frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{touch}\ e : \texttt{unit}}$$

$$\text{[T-MKDIR]} \quad \frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{mkdir}\ e : \texttt{unit}}$$

$$\text{[T-RM]} \quad \frac{\Gamma \vdash e : \texttt{filepath}}{\Gamma \vdash \texttt{rm}\ e : \mathit{unit}}$$

Subtyping

$$\begin{aligned} \texttt{bool} &<: \texttt{string} \\ \texttt{num} &<: \texttt{string} \\ \texttt{filepath} &<: \texttt{string} \\ \texttt{unit} &<: \texttt{string} \\ \lfloor \tau \rfloor &<: \texttt{string} \end{aligned}$$

$$\text{[T-SUB]} \quad \frac{\Gamma \vdash e : \tau_s \quad \tau_s <: \tau}{\Gamma \vdash e : \tau}$$

$$\text{[T-REFL]} \quad \frac{}{\tau <: \tau}$$

$$\text{[T-TRANS]} \quad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

$$\text{[T-ARROW]} \quad \frac{\tau_{21} <: \tau_{11} \quad \tau_{12} <: \tau_{22}}{\tau_{11} \to \tau_{12} <: \tau_{21} \to \tau_{22}}$$

Quoting

$$\lfloor \lfloor \tau \rfloor \rfloor = \lfloor \tau \rfloor \quad \text{(T-IDEM)}$$

$$\lfloor string \rfloor = string \quad \text{(T-QSUB)}$$

$$string \circ string = string \quad \text{(T-CONSTR)}$$

$$\lfloor \tau_1 \rfloor \circ \lfloor \tau_2 \rfloor = \lfloor \tau_1 \circ \tau_2 \rfloor \quad \text{(T-QUO)}$$