The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Engineering, Computing and Cybernetics (CECC)

Sprinkle Magic on the Dance: Enriching a Verified Choreographic Language with a Simply Typed Lambda Calculus

— Honours project (S1/S2 2024)

A thesis submitted for the degree
Bachelor of Advanced Computing (Research and Development)

By:
Xin Lu

Supervisor:
Dr. Michael Norrish

October 2024

Declaration:

I declare that this work:

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

October, Xin Lu

Acknowledgements

If you wish to do so, you can include some Acknowledgements here. If you don't want to, just comment out the line where this file is included.

There is absolutely no need to write an Acknowledgement section, so only do so when you *want* to – it's always important to stay sincere. One reason for including an acknowledgement could be to thank your supervisor for extraordinary supervision (or any other reason you deem noteworthy). Some supervisors sacrifice a lot, e.g., are always available, meet on weekends, provide multiple rounds of corrections for theses reports, or the like (keep in mind that writing a thesis is special for you, but not for them, so they do actually not have any reason to sacrifice their private time for this!). Seeing acknowledgements in this report can feel like a nice appreciation of this voluntary effort. For large works that form the end of some studies (like an Honours or Master thesis), it is also not uncommon to read acknowledgements to one's parents or partner. But again, completely optional!

Abstract

Distributed systems are ubiquitous but writing endpoint programs can be error-prone since mismatched message sending and receiving can lead to errors such as deadlock, where the system indefinitely awaits a message. Choreography offers a solution by providing a global description of how messages are exchanged among endpoints, where message mismatches are disallowed — a property called "deadlock-free by design." The global choreography is then projected into process models for each endpoint via EndPoint Projection (EPP), preserving the deadlock-free property.

While many choreography languages focus on message exchange behaviors, few address the local computations occurring within endpoints. Most current languages assume local computation results or delegate them to external languages. While this offers a reasoning ground for studying message exchange behaviours of choreography, when it comes to writing a concrete choreography program, the former can only exchange literal values and the latter leads to cumbersome code due to the addition of an external computation program which typically involves conversions between choreography values and external data types.

Hence in this thesis, we extend Kalas, a state-of-the-art choreography language with verified end-to-end compilation, with a local language Sprinkles, such that local computations are handled gracefully within a few lines of codes. Moreover, it also allows us to formally analyse the message exchange behaviours of choreography when local computations are considered.

We design Sprinkles as a simply typed lambda calculus with function closure. We use a functional bis-step semantics with clocks to ensure the evaluation function for Sprinkles is total. We prove type soundness for the proposed semantics and typing rules. We also provide a strong normalisation proof for Sprinkles.

We extend the Kalas' *let* transition with Sprinkles expressions. Besides common data types such as integer, string, and boolean, we also add function, pair, and sum types to the local computation in choreography . Common operators for our data types are included, such as addition, modulo, and negation. An integer-string converter is implemented as well to handle message strings in Kalas. Last but not least, we prove the enriched Kalas enjoys progress. We also show type preservation holds for non-recursive, synchronous transitions.

Table of Contents

Chapter 1

Introduction

Distributed systems consist of multiple endpoints that communicate by exchanging messages, operating asynchronously and in parallel as these messages are sent and received among the various endpoints. However, programming distributed systems is notoriously error-prone because programmers must implement the communication protocol by developing individual endpoint programs. Mismatched message sending and receiving can lead to errors such as *deadlock*, where the system indefinitely waits for a message.

Choreographic programming arises as a formalisaton to address this issue by providing a concrete global description of how the messages are exchanged between endpoints in a distributed system. A choreography program is written in a similar style to the "Alice and Bob" notation by Needham and Schroeder (1978):

1. $\texttt{Alice} \rightarrow \texttt{Bob} : \mathit{key}$
2. $\texttt{Bob} \rightarrow \texttt{Alice} : \mathit{message}$

Thus message mismatches are disallowed from the choreographic perspective. This property is known as *deadlock-free-by-design*. The global choreography can then be projected into process models for each endpoint via EndPoint Projection (EPP), with properties such as deadlock free by design preserved (Hallal et al., 2018).

When analysing message exchange behaviors in choreography, such as multiparty sessions, asynchrony, and parallelism, it is reasonable to assume good behaviors in local computation (Montesi and Yoshida, 2013; Cruz-Filipe and Montesi, 2017; Carbone and Montesi, 2013). Consequently, most choreography languages focus primarily on message exchange behaviours, with few paying attention to the local computation occurring in individual endpoints. Although Hirsch and Garg (2022) demonstrates that for their assumed local language, as long as it exhibits type preservation and progress, the choreography inherits these properties, this result may not hold true for all local languages.

Additionally, when it comes to writing the choreography program that implements concrete system behaviours, if local computation is ever required by the system, one must describe the inputs and outputs of local computations. Current work either delegate this part to an assumed well-behaved external implementation, for example, Kalas Pohjola et al. (2022) and Pirouette Hirsch and Garg (2022), or provide a basic framework where only Church numerals are considered, as

in Core Choreography (CC) Cruz-Filipe and Montesi (2017). This makes writing choreography program with local computation implementation an unpleasant experience. For instance, a choreography program in Kalas, where the client computes modulo locally using input from the server and then sends the result back, will look like:

Example 1.0.1 (Local computation — Modulo)**.**

Kalas	External computation
1. `server`.*var* -> `client`.x;	**fun** *mod* x =
2. **let** v@`client` = `mod`(x) **in**	case Option.map (fn s ⇒ valOf (Int.fromString s)) (hd x) of
3. `client`.v -> `server`.*result*;	None ⇒ None
	| Some n ⇒ Some [Int.toString (n MOD y)]

In Kalas, processes communicate exclusively via strings. Therefore, in the external implementation of `mod`(x), the string value of x must first be converted to a number. Then client computes the modulo on the converted input. Afterward, the result is converted back to a string before updating the client process variable v. We can easily see from this example that how data type conversions between choreography and external language lead to cumbersome code.

Thus this thesis extends Kalas, a verified choreography language with machine-checked end-to-end compilation, by introducing Sprinkles, a simple language of expressions over types such as integers, strings and booleans. Using the extended Kalas, local computations can be handled gracefully within a few lines of code. The previous choreography where client computes the modulo of input integer can be written as in Example 1.0.2.

By providing a concrete local language syntax and semantics, we are able to formally anlayse the message exchange behaviours of choreography in terms of progress and type preservation when local computations are considered. We show that type soundness and strong normalisation properties of Sprinkles lead to progress for the enriched Kalas. Our semantics and typing system also allow us to show type preservation for non-recursive and synchronous transitions in the enriched Kalas.

Example 1.0.2 (Local computation with Sprinkles — Modulo)**.**

1. `server`.*var* -> `client`.x;
2. **let** v@`client` = StrOf ((NumOf (Var x)) Mod (Var y)) **in**
3. `client`.v -> `server`.*result*;

To summarise, this thesis provides three main contributions.

- The first contribution is Sprinkles, a simply typed lambda calculus with rich types. Following the approach taken by Owens et al. (2016), we use a functional big-step semantics with environments and closures to ensure the evaluation function for Sprinkles' expressions is total. Our evaluation strategy is call-by-value. We also provide a typing system and give type soundness proof for the proposed semantics and typing rules. Sprinkles is implemented using the higher-order logic proof assistant HOL4 (Slind and Norrish, 2008) and all the proofs are conducted within HOL4.
- The second contribution is the strong normalization proof for Sprinkles. We follow the standard practice for proving strong normalization in simply typed lambda calculus, with a specific case for function closure in the definition of the strong normalization relation.

- The third contribution is Kalas enriched with Sprinkles. Besides common data types such as integer, string, and boolean, we also add function, pair, and sum types to the local computation in Kalas. Common operators for our data types are included, such as addition, modulo, and negation. Integer-string conversion is implemented as well for the integration with Kalas. Last but not least, we prove the enriched Kalas enjoys an important property: *progress*. We also show type preservation holds for non-recursive, synchronous transitions in the enriched Kalas.

Chapter 2

Background

2.1 Choreographic Programming

A distributed system is defined as a collection of autonomous computing elements that operate according to users' expectations (Mullender, 1990). Message exchanges are at the core of distributed system design, as they enable nodes to collaborate and share resources. Without such exchanges, there would no need to connect different nodes within a single network. In a traditional approach to describe a distributed system, one must provide a detailed specification of the operations at each individual node. For example, communication between node A and node B involves A sending a message and B receiving it. However, this approach can lead to issues such as mismatched message sending and receiving, potentially resulting in deadlocks or race conditions.

Choreography emerges as an effort to eliminate message mismatches in distributed systems by offering a global description of how messages should be exchanged. This method is similar to dance choreography, where the focus is on the overall coordination of steps and movements for the entire performance, rather than detailing the control points of individual dancers.

While it is exciting to eliminate mismatched messages from the choreography perspective, choreography cannot be executed directly on individual nodes within a system. As a result, implementing endpoint programs is still necessary. This led to the idea of EndPoint Projection (EPP), which involves projecting a choreography into endpoint programs so that each endpoint correctly implements the behaviours defined by its role in the choreography. This concept was first outlined in the design document for the Web Services Choreography Description Language (WS-CDL) (W3C WS-CDL Working Group, 2005), and was subsequently formalised into the theory of EPP by Carbone et al. (2007). In this context, two key properties are established: *soundness* and *completeness*. Soundness ensures that all projected endpoint communications conform to the choreography description, while completeness guarantees that all communications specified in the choreography are reflected in the projected endpoint codes.

2.2 Interactive Theorem Proving

Higher Order Logic (HOL) is a programming language developed on top of the metalanguage ML (Gordon, 1988). ML is an interactive programming language that allows users to evaluate expressions, with the corresponding values and types being printed. Users can also perform declarations to bind values to names. In HOL, terms are represented as an ML type called *term*, while the types of HOL terms are represented as an ML type called *type*.

We are mainly interested in using HOL as a proof assistant. HOL supports proofs by natural deduction. In this context, a sequent in HOL is defined as a pair (Γ, t), where Γ represents a set of assumptions and t denotes the conclusion formula. A theorem in HOL is a sequent that is either an axiom or a sequent derived from other theorems using rules of inference.

A key feature of HOL as a proof assistant is its support for *goal-oriented* proving through the use of tactics. In a goal-oriented proof, the goal is broken down into subgoals through justifications. This process continues until the subgoals become straightforward to prove. A *tactic* is an ML function that reduces the current goal into subgoals. For example, the conjunction tactic takes a goal of the form $p \wedge q$ and breaks it into two subgoals: p and q.

HOL includes many built-in libraries that cover useful theories for option types, natural numbers, set operations, and more. These theories can be loaded into the current working file, allowing users to build definitions and proofs based on the imported theories. Consequently, we develop Sprinkles and its related theorems by first creating HOL definitions for our syntax, functions, and relations based on the imported theories. We then apply HOL tactics to prove the stated theorems in a goal-oriented manner. The most frequently used theory for this project, aside from common theories like natural numbers and set operations, is the finite map theory. Both Kalas' states and Sprinkles's environments are implemented as finite maps. This project is also based on HOL4, the latest supported version of HOL (Slind and Norrish, 2008).

2.3 Kalas

Kalas is a state-of-the-art choreography language featuring verified end-to-end compilation. Its syntax is outlined in Definition 1 and is similar to that of Core Choreography (CC) (Cruz-Filipe and Montesi, 2017). Kalas supports recursions and sequencing. Besides the synchronous transitions, Kalas' transition rules also handle parallel and asynchronous transitions (Definition 2). Most importantly, a compiler with an end-to-end, machine-checked proof of correctness for Kalas is provided.

The compilation process for Kalas consists of several phases, during which the top-level choreography is projected into endpoint process algebra. The process programs are further compiled phase by phase to address message sizes and the Kalas fixpoint operator. The compiler reaches CakeML (Kumar et al., 2014), a functional programming language with verified compilation, in the final compilation phase, which gives Kalas semantics preservation down to the machine code. Soundness and completeness proofs are provided for each compilation phase to ensure operational correctness at every stage. As a result, Kalas enjoys network-level deadlock free and correctness.

Kalas is implemented using HOL4, with all its proofs conducted through the HOL proof assistant. This foundation leads us to implement Sprinkles and its associated proofs using HOL.

Definition 1 (Kalas syntax)**.** Choreographies *in Kalas, ranged over by C, are inductively defined by the grammar*

$$
\begin{array}{lllll}
C & ::= & p_1.v_1 \twoheadrightarrow p_2.v_2; C & \textit{(com)} & p_1 \twoheadrightarrow p_2[b]; C \quad \textit{(sel)} \\
 & & \textbf{if } v@p \textbf{ then } C_1 \textbf{ else } C_2 & \textit{(if)} & \textbf{let } v@p = f(\tilde{v}) \textbf{ in } C \quad \textit{(let)} \\
 & & \mu X.\ C & \textit{(fix)} & X \quad \textit{(var)} \\
 & & \mathbf{0} & \textit{(nil)} &
\end{array}
$$

Definition 2 (Parallel and asynchronous transitions in Kalas)**.**

$$\textsc{Com-S}\ \frac{s \rhd C \xrightarrow[l]{\alpha} s' \rhd C' \quad p_1 \notin \mathrm{fp}(\alpha) \quad p_2 \notin \mathrm{fp}(\alpha)}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; C \xrightarrow[l]{\alpha} s' \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; C'}$$

$$\textsc{Com-A}\ \frac{s \rhd C \xrightarrow[l]{\alpha} s' \rhd C' \quad p_1 \in \mathrm{fp}(\alpha) \quad \mathrm{wv}(\alpha) \neq (v_1, p_1) \quad p_2 \notin \mathrm{fp}(\alpha)}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; C \xrightarrow[(p_1.v_1 \twoheadrightarrow p_2.v_2)::l]{\alpha} s' \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; C'}$$

Chapter 3

Related Work

3.1 Choreography Models

3.1.1 Typing System for Choreography

Choreography language model can ensure deadlock freedom without using a typing system. The property is typically proven through structural induction on the choreography's semantics, where an applicable rule always enables reduction, or reduction follows by inductive hypothesis (Carbone and Montesi, 2013; Cruz-Filipe and Montesi, 2017; Pohjola et al., 2022). Typing system in this case might still be desired since it can discipline choreography in other ways such as correct protocol implementation.

Channel Choreography (ChC) by Carbone and Montesi (2013) is a rich choreography language where a choreography program consists of where a program consists of roles, threads, and sessions that implement communication protocols. Deadlock freedom is guaranteed by its semantics, while the typing system ensures correct protocol implementation by sessions. The typing context includes three components: a service environment Γ, which stores global types for public channels specifying session execution and local expressions (annotated with threads); a thread environment Θ, which tracks the roles of threads in each session; and a session environment Δ, which stores the types of active sessions.

It ensures that a well-typed choreography, with public channels specified by Γ and threads assuming roles in Θ, maintains disciplined sessions governed by Δ. Additionally, runtime typing introduces a delegation environment to handle changes in typing context due to asynchrony or parallelism, ensuring the program adheres to the protocol during execution.

When local computation is involved in the choreography, since we cannot solve the halting problem in semantics by deciding whether the local computation will terminate, a typing system is desired to reassert deadlock freedom. To the best of our knowledge, the closet work to this discussion is Pirouette by Hirsch and Garg (2022). They assume a substitution model with small-step semantics for the local language. Based on a set of admissible typing rules for the sake of providing a reasoning ground, their results show that type preservation and progress in the local language ensure the same for the choreography language. Their progress result aligns with our

results, but our local language adopts big-step semantics and thus preservation for choreography requires strong normalisation from local language rather than type preservation.

Pirouette is a higher-order functional choreography language with three value types: local values, local functions (mapping local values to choreographies), and global functions (mapping choreographies to choreographies). This structure enables choreographies to return values, forming the basis of its typing system. On the other hand, since neither Kalas or the enriched Kalas has return values, our typing system mainly checks for the well-formedness of a choreography within the typing environment and if any local computation is involved, we discipline it with its own typing system in a localised typing environment. Pirouette types its local values in a similar projected typing environment, binding variables to types at a specific location.

Another state-of-art functional choreography language Chorλ by Cruz-Filipe et al. (2022) uses a different approach than Pirouette, where choreographies are interpreted as terms in λ-calculus. Chorλ assumes local values for communications, without focusing on how they are computed. This results in a distinct typing system: local value types are annotated with roles and are part of the global types, rather than being projected from a global environment. Type preservation and progress follow from Chorλ's typing and semantic rules.

3.1.2 Handling Exceptions

- Choral: an object-oriented choreography implemented as a java library (Giallorenzo et al., 2024). It treats choreography as class and EPP as generating role classes from the choreography implementation. It also offloads the local computation to Java. Higher-order functional choreography models such as Pirouette and Chorλ can be viewed as formal model candidates for it. Its type checker is similar to the typing system in ChC where it has types for public channels where roles for sender and receiver are specified as well as the type of messages being communicated an it also has local types annotated with roles.

In terms of communication failure which may raise an exception when a role is trying to perform operation based on reading from the place where a received message has not arrived yet, Choral implements the failure model in RC by Montesi and Peressotti (2017), using recover strategies such as capped attempts or timer within java try-catch block.

- RC, a model where communication failure is considered, recover strategy for senders and receivers, either while loop until received (exactly once delivery for setting 1), or with a timer or capped tries (best efforts); typing system to ensure almost once delivery (and exactly one delivery where message won't be lost); they use configuration where sender and receiver have stacks and it is initially false, with payload value, or ticked meaning no longer in the stack (sent/received) to implement send/received and the failure rules for semantics and typings; we do not consider message sending failure, but we do have exception caused by local computation failure (e.g. division by zero) or bad message value type (not a string), and we record the exception using transition labels. But we do not consider any recover strategy and always transits the choreography into a termination state (nil)

3.2 Approach to Strong Normalisation

Though Sprinkles is an extension of simply typed lambda calculus (STLC), its functional big-step semantics and function closure makes its strong normalisation proof slightly different to a standard one.

A standard strong normalisation proof for STLC with non-deterministic reductions, small-step semantics, and substitutions would have to define an inductive notion SN for strong normalisation and prove the notion SN is sound that a term of SN is truly strongly normalising, as we see in the proof given by Abel et al. (2019).

The definition of SN also has to handle application of terms by introducing an extra constrain called strong head reduction, since $[N/x]M$ being strongly normalising does not imply $(\lambda x : A.M)N$ or N are also strongly normalising since there may be no occurrences of x in N. Thus the strong head reduction requires N to have the notion of SN as well.

However, Sprinkles uses a functional big-step semantics outlined by Owens et al. (2016), which leads us to implement our semantics as an evaluation function with clocks. The functional big-step semantics also ensures the evaluation function is total and deterministic.

As a result, strong normalisation proof for Sprinkles is equivalent to a weak one, and it does not require us to consider the reduction sequences nor need to consider the strong head reduction for application when using a substitution model. Our proof, is more closed to the strong normalisation proof for a simpler call-by-value STLC in HOL repository by Owens and Laurent (2016), in a sense that they both adopt a functional big-step semantics and use an environment to store bindings for free variables, but Sprinkles has richer data types compared to the STLC in HOL example.

Chapter 4

Sprinkles: A Simply Typed Lambda Calculus

We introduce Sprinkles, a language over expressions that closely resembles a simply typed lambda calculus. We adopt the functional big-step semantics described by Owens et al. (2016), which guarantees that our evaluation function is both total and deterministic. Our implementation includes an evaluation function that utilises clocks to evaluate expressions, yielding either values or exceptions. We also account for common exceptions, such as division by zero and improperly formatted strings during integer-string conversions.

We use an environment, denoted as E, to store values for free variables. Thus we need to pass E as an argument to our evaluation function. While a substitution model may be mathematically appealing, our environment-based approach aligns well with functional big-step semantics and is relatively easy to implement. This setup leads us to define a typing environment, G, which stores types for free variables. Consequently, expressions in Sprinkles are always typed within some typing environment.

When it comes to formalising the type soundness statement for Sprinkles, we want to ensure that a typed expression is evaluated in a *correct* environment E according to the typing environment G that types the expression, rather than an arbitrary one. To achieve this, we define $\mathsf{envtype}\ G\ E$ to characterise such a *correct* environment E in relation to a typing environment G.

4.1 Syntax

Definition 3 (Sprinkles syntax)**.** Values *and* expressions *in Sprinkles, ranged over by* v, e*, are inductively defined by the grammar*

$$
\begin{array}{rcl}
v & ::= & \mathsf{IntV}\ n \mid \mathsf{StrV}\ s \mid \mathsf{BoolV}\ b \mid \mathsf{Clos}\ s\ e\ E \mid \mathsf{PairV}\ v_1\ v_2 \mid \mathsf{SumLV}\ v \mid \mathsf{SumRV}\ v \\
\mathit{bop} & ::= & \mathsf{Add} \mid \mathsf{Concat} \mid \mathsf{Mult} \mid \mathsf{Div} \mid \mathsf{Mod} \mid \mathsf{Less} \mid \mathsf{And} \mid \mathsf{Or} \mid \mathsf{Eq} \mid \mathsf{Sub} \mid \mathsf{Pair} \\
\mathit{uop} & ::= & \mathsf{Not} \mid \mathsf{NumOf} \mid \mathsf{StrOf} \mid \mathsf{Fst} \mid \mathsf{Snd} \mid \mathsf{SumL} \mid \mathsf{SumR} \\
e & ::= & \begin{array}{llll}
\mathsf{Var}\ x & \mathit{(var)} & \mathsf{StrLit}\ s & \mathit{(str)} \\
\mathsf{IntLit}\ n & \mathit{(int)} & \mathsf{BoolLit}\ b & \mathit{(bool)} \\
\mathsf{BinOp}\ \mathit{bop}\ e_1\ e_2 & \mathit{(bop)} & \mathsf{Uop}\ \mathit{uop}\ e & \mathit{(uop)} \\
\mathsf{If}\ \mathit{bg}\ e_1\ e_2 & \mathit{(if)} & \mathsf{Let}\ x\ e_1\ e_2 & \mathit{(let)} \\
\mathsf{Fn}\ x\ e & \mathit{(fn)} & \mathsf{App}\ e_1\ e_2 & \mathit{(app)} \\
\mathsf{Case}\ e\ x\ e_1\ y\ e_2 & \mathit{(case)} & &
\end{array}
\end{array}
$$

$\mathsf{Var}\ x$ represents a variable where x is the string that represents variable's name. $\mathsf{BinOp}\ \mathit{bop}\ e_1\ e_2$ represents a binary operation and $\mathsf{Uop}\ \mathit{uop}\ e$ represents unary operation. We include literals for string, integer and boolean in our syntax to allow expressions such as $\mathsf{BinOp\ Add}\ (\mathsf{Var}\ x)\ (\mathsf{IntLit}\ 1)$. If, Let, Fn, and App are standard. $\mathsf{Case}\ e\ x\ e_1\ y\ e_2$ represents a sum case expression, where the bound variable x and expression *e1* correspond to the left branch, while the bound variable y and *e2* correspond to the right branch. Free variables for Sprinkles' expressions are standard.

We separate the syntax definitions for binary and unary operations from the concrete definitions of specific binary and unary operators. This makes it easy to extend the set of operators: to add a new operator, we only need to update the *bop* or *uop* definitions and provide the corresponding evaluation rule. This approach also significantly reduces the number of cases in proofs that depend on the syntactic structure of expressions, as there's just a single case for all binary operators and a single case for all unary operators.

We include exceptions to handle division by zero and bad formatted integer strings. They are thrown by the corresponding operators. We define three observable evaluation results: as either values, or we encounter a type error (for example, $\mathsf{BinOp\ Add}\ (\mathsf{IntLit}\ 1)\ (\mathsf{StrLit}\ \text{“1”})$), or we reach an exception, or the evaluation times out for the given clock value.

We include exceptions to handle cases like division by zero and improperly formatted integer strings, which are raised by the relevant operators. We define four observable outcomes for evaluation: a successful value v, a TypeError for evaluating expressions like $\mathsf{BinOp\ Add}\ (\mathsf{IntLit}\ 1)\ (\mathsf{StrLit}\ \text{“1”})$, an exception $\mathsf{Exn}\ \mathit{exn}$, or a Timeout if the evaluation exceeds the specified clock limit.

4.2 Statics

Our typing rules for expressions closely follow those of a simply typed lambda calculus. Because values are not part of the expression syntax, we also define a separate typing relation for values.

Our expressions are typed in a typing environment G which stores mappings from variable names to types. Similar to a variable expression is evaluated by looking up variable name in the given environment E, a variable expression is typed by looking up type stored under the variable name in the given typing environment G. This setup, requires us to constrain the typing environment G and dynamic environment E in the type soundness statement to make sure we are evaluating the typed expression using a correct dynamic environment that aligns with the typing constrain specified in G.

Our expressions are typed within a typing environment, G, which maps variable names to types. Just as a variable expression is evaluated by looking up its variable name in the dynamic environment E, a variable expression is typed by checking the type associated with that variable name in the typing environment G. This setup requires the dynamic environment E to be aligned

with the types specified in the typing environment G in the type soundness statement, so we can ensure expressions are evaluated in a correct environment. We formalise this constrain in Section 4.2.1.

For typing binary and unary operations, we delegate the task of checking operand types against a specific operator to two external typing relations, boptype and uoptype. This approach allows us to define just two general rules for binary and unary operations when specifying typing rules for expressions. This results in fewer cases in the type soundness proof for operators, requiring only two separate soundness lemmas for boptype and uoptype. Our design showcases a clear separation of concerns.

4.2.1 Syntax and Typing Rules

Definition 4. - *types*
Definition 5. - *typing relations: typecheck*

- typing relations: typecheck, value_type, uop_type, bop_type

Expressions

Typing statements for expressions are formulated as $G \vdash_s e : t$, where t is the type of expression e given the typing environment G. A variable x is typed by looking it up in G. The typing rules for other expressions are standard.

To type a binary operation BinOp *bop* e_1 e_2, we need the types of *e1* and *e2*, given G, to satisfy the constraints defined by boptype for the operator *bop*. The typing rules for unary operations are defined in a similar manner.

Environments

Before we define types for values, we first introduce envtype G E as a typing constraint that relates the typing environment G to the dynamic environment E. Specifically, envtype G E requires that the values of variables stored in E conform to the types of variables specified in G.
Definition 6. - *envtype definition.*

Values

Definition 7. - *typing relations: value_type*

In $\vdash_s v : t$, a value v is related to a type t. Literal value cases are obvious. Typings for pair and sum values are standard. Closure value, represented as Clos s e E, is a triple (s, e, E) corresponding to a function expression Fn s e. In this closure, s represents the function parameter, e is the function body, and E is the environment that stores the bindings of free variables when the function definition is evaluated.

To type a closure value Clos s e E, if G accurately describes the types of variable values stored in E, we require that e has the output type *t2* in $G[s := t_1]$. The constraints for G and E align with Definition **??**. [1]

[1] We do not apply envtype G E directly in the closure case because the definition of envtype depends on $\vdash_s$, and we want to avoid mutually recursive definitions between a definition and a relation in HOL4.

Requiring the constraint envtype G E in the definition of $\vdash_s$ Clos s e E : fnT t_1 t_2 is appropriate because, when the closure value is applied, e will be evaluated in the extended closure environment $E[s := v]$ for some argument value v. Therefore, we want to have e typed within a typing environment that accurately describes the types of values stored in $E[s := v]$. As we will see in Section **??**, the constraint envtype G E is also necessary for the proof of the App e_1 e_2 case in the type soundness proof.

As a consequence of Definition 7 and 5, SumLV v does not have a unique type because *t2* is arbitrary, and the same applies to SumRV v. Furthermore, the function Fn s (Var s) also has non-unique types, as the variable s can take on arbitrary types.

4.2.2 Typing Examples

Example ? discussed in Section ? to show our function closure supports currying can be typed using the rules in Definition ?. A derivation tree is provided in Appendix ?.

However, the recursion program (Example ?) discussed earlier cannot be typed. Since it is a let expression, the last rule applied must be T-let. We can type f 1 in the extended typing environment (f:int-¿int). But we cannot type the definition (fn ... some complex expression) in the empty typing environment since f is free in the definition expression. A stuck derivation tree is provided in Appendix ?.

4.3 Dynamics

We adopt a functional big-step semantics. As a result, our semantics are implemented as an evaluation function that takes a clock value c, an environment E, and an expression e as inputs. We use clocks since the size of expressions are decreasing except for the application case, so decreasing the clock value when evaluation an application ensures the termination of our evaluation function. Our definition of evaluation function also implies a call-by-value evaluation strategy.

We define the evaluation function as:
Definition 8. - *semantics rules*

Rules for If, Let, and Case are standard. We evaluate Var x by looking up the variable x in the provided environment E. If E does not contain a binding for x, it indicates that x is an undefined variable. Thus TypeError is returned.

For *Bin bop* e_1 e_2, we first evaluate *e1* and *e2* to obtain *v1* and *v2*. Next, we use *eva_bop bop* v_1 v_2 to compute the result based on the specific rule for the operator *bop*, which is defined in eval_bop. The evaluations for unary operations are defined in a similar manner. The definitions of eval_uop and eval_bop can be found in Appendix ?.

Based on the rules outlined so far, let's consider an evaluation example of the expression Uop NumOf (Var x). The function eval_exp c E (Uop NumOf (Var x)) will first compute eval_exp c E (Var x) by looking up the variable x in the environment E. If x is not found in E, the function will return TypeError. If the lookup succeeds, eval_exp c E (Var x) will return a value v, and we will then compute eval_uop NumOf v. If v is not a correctly formatted integer string (for instance, if it contains a non-numeric character), eval_uop NumOf v will return Exn BadStr. Otherwise, eval_uop NumOf v will return the converted integer value IntV n. The relevant format checking and string conversion algorithms are utilised by eval_uop during the evaluation process.

4.3.1 Function Closure

We first define $f \lfloor_s$ as:
Definition 9. - *definition of* DRESTRICT

Then we give the evaluation rule for Fn s e:
Definition 10. - *function semantics rule (restricted version)*

However, our rule stores a restricted environment Clos s e $(E \lfloor_{\text{fv}(e)} \setminus\setminus s)$ in the evaluated closure, which may seem unexpected. Nonetheless, we argue that Clos s e $(E \lfloor_{\text{fv}(e)} \setminus\setminus s)$ behaves the same as Clos s e E during evaluations and will produce the same result.

First, $E \lfloor_{\text{fv}(e)}$ includes values for all free variables in e. Second, excluding s is safe because when we apply the restricted closure to a value, we always bind s to the argument value. So the previous value of s stored in the environment becomes irrelevant. This modification is also necessary to preserve Proposition 3. Further discussions can be found in Section 6.6.1.

By using a closure, we can keep track of the values for bound variables in nested function definitions. If we do not record the environment when a function definition is evaluated, we risk losing the bindings for all other bound variables from outer scopes once we complete the evaluation of the innermost function definition. Additionally, currying would be impossible without utilising a closure, as illustrated in Example 4.3.1.

Example 4.3.1. - example: failure without closure

When we are evaluating Fn y (Add (Var x) (Var y 1)) in line 5 we know x is bound to 1. But we lose this information when we choose $(y,$ Add (Var x) (Var y)) to be the evaluation result. So we only know y is bound to IntLit 2 when we try to evaluate Add (Var x) (Var y 1) in line 7.

By using function closure to record values of all bound variables when a function definition is being evaluated, we are able to evaluate Add (Var x) (Var y 1), as illustrated in line 7 of Example 4.3.2. It is possible that a function body contains unbound free variables, but such expression cannot be typed in an empty environment and it is reasonable to make its evaluation fails. Discussions on typing for function closure are in Section ?.

Example 4.3.2. - restricted currying evaluation example

Recursion

Sprinkles's syntax supports recursion, as one can write a Y combinator However, Let f e_1 e_2 does not support recursion because f is not bound within *e1*. Additionally, our typing system is unable to type the Y combinator.

We propose that recursion can be easily and effectively integrated into Sprinkles by introducing the syntax Letrec. The expression Letrec f *s e1 e2* will be evaluated into a quadruple (f, x, e, E). This quadruple is similar to the one described by Hardin et al. (2021).

4.3.2 Main Properties

Functional big-step semantics ensures that our evaluation function is total and deterministic by design. With this in mind, we define the evaluation of an expression as *terminating* if there exists a clock value for which the evaluation function returns a result that is not Timeout. Conversely,

if the evaluation returns Timeout for all clock values, we say that the evaluation of the expression *diverges*.

The evaluation of Sprinkles is also deterministic in the sense that if the evaluation for an expression terminates with an initial clock value, then increasing the clock will always yield the same result. We demonstrate this result as a corollary that follows from Lemmas 1, 2, and 3.

The first lemma states that if the expression e evaluates to a value v in the environment E with a clock value *c0*, then evaluating the same expression in the same environment with a larger clock value *c1* will also yield the same value v. The statements for Lemmas 2 and 3 are analogous.

Lemma 1.

$$\begin{aligned} &\mathsf{eval\_exp}\ cl_0\ E\ e\ =\ \mathsf{Value}\ v\ \Rightarrow \\ &\forall\, cl_1.\ cl_1\ \geq\ cl_0\ \Rightarrow\ \mathsf{eval\_exp}\ cl_1\ E\ e\ =\ \mathsf{Value}\ v \end{aligned}$$

Proof: Using the well-founded lexicographic induction principle by Definition 8, results for each case directly follows the inductive hypotheses. The induction principle can be found in Appendix ?. □

Lemma 2. - *clock_exn_increment*

Proof: The proof relies on the same induction principle for Lemma 1 and depends on Lemma 1, since in the inductive case an exception can occur after a value. □

Lemma 3. - *clock_typeerr_increment*

Proof: Similar to the proof for Lemma 2. □

Corollary 1. - *clock_increment*

Proof: Cases analyses based on the syntax form of r. Each case follows either Lemma 1, 2, or 3. □

Corollary 1 is important for the proof of strong normalisation. For example, in the case of the if statement, we know that the evaluation of the boolean guard and the two branches terminates by the inductive hypotheses for their respective clocks. We need to ensure that there exists a common clock value on which the evaluation of all three expressions also terminates and returns the same results. Further discussion on the use of Corollary 1 in the strong normalisation proof can be found in Section 5.3.2.

4.4 Type Soundness

4.4.1 Lemmas

We will discuss several lemmas and propositions for the proposed semantics and typing rules before proceeding with the type soundness proof.

Our type soundness proof is similar to the type soundness proof for a simply typed lambda calculus by Pierce (2002) in a sense that we require, or at least require an equivalent version of, the generation lemma, the lemma of canonical forms of types, and the type preservation under substitution lemma used in Pierce (2002)'s proof.

Our type soundness proof is similar to the type soundness proof for a simply typed lambda calculus with small-step semantics and the substitution model, such as the one presented by Pierce (2002). Specifically, we require, or at least need an equivalent version of, the generation lemma, the lemma on canonical forms of types, and the lemma on type preservation under substitution, all of which are utilised in Pierce (2002)'s proof.

Lemma 4. - *typecheck_*_thm: the generation lemma*

Proof: Immediately follows from Definition 5. □

The generation lemma enables us to advance one step in the type derivation tree from the bottom to the top. It allows us to obtain types and typing information for the sub-expressions within a well-typed expression.

Lemma 5. - *valuetype_EQ_*: the lemma of canonical forms of types (only useful cases)*

Proof: Immediately follows from Definition 11. □

The canonical form lemma provides the concrete syntax of a value v along with the typing information for the expressions involved in that concrete syntax, given that we have $\vdash_s v : t$. This is particularly useful for inductive cases in the type soundness proof, as the inductive hypothesis tells us that expressions evaluate to a value v and that $\vdash_s v : t$ holds. Moreover, the concrete syntax of v always aligns with the expressions in goal.

The type preservation under substitution lemma used by Pierce (2002) states that substituting variables with terms of appropriate types preserves the original type. Since we evaluate variables by looking them up in an environment E, the concept of envtype G E (as defined in Definition 6) is analogous to the type preservation under substitution lemma.

As discussed in Section 4.2.1 and Section 4.3, we first evaluate the argument expressions into values, and then we delegate the evaluation of binary and unary operations to eval_bop and eval_uop. Similarly, we first type the argument expressions and then check the input and output types against the concrete operators using separate typing relations, boptype and uoptype. This approach leads to two important properties: if the argument values are properly typed and the operator types match the argument types, then eval_bop and eval_uop will always output a value of the correct type or result in an exception. We refer to these properties as type soundness for eval_bop and eval_uop.

Proposition 1. - *bop soundness*

Proof: boptype is non-recursive. We prove by performing cases analyses based on boptype rules and each case follows from Lemma 5. □

Proposition 2. - *uop soundness*

Proof: Same as the above. □

Proposition 1 and Proposition 2 are required for the binary and unary operator cases in the type soundness proof. By the inductive hypothesis, we will have well-typed evaluated argument values, and we need to show that eval_bop or eval_uop either outputs a well-typed value or results in an exception.

There is also a lemma that allows us to update G and E with a typed value and re-establish the envtype property:

Lemma 6. - *envtype_lemma*

Proof: Since the typing rules for values (Definition 7) are defined inductively, we will prove them by induction on Definition 7. The base cases include integer, string, and boolean literals, which directly follow from the definition of envtype (Definition 6). The cases for pair and sum values derive directly from the inductive hypothesis. The closure value case does not rely on the inductive hypothesis; instead, it follows from the closure case in the lemma of canonical forms of types (Lemma 5). □

This lemma is useful in the function, application, and sum cases of the type soundness proof, where we have sub-expressions within the goal expression that are typed in an extended typing environment. We need to re-establish the envtype property for both the extended typing environment and the dynamic environment so that we can apply the results from the corresponding inductive hypothesis.

Restricted Function Closure

Since we evaluate a function expression into a restricted closure value (as discussed in Section 4.3.1), we will need the following lemmas for the function case in the type soundness proof. Given that $G \vdash_s \mathsf{Fn}\ s\ e : (\mathsf{fnT}\ t_1\ t_2)$, Lemma 4 ensures that $G[s := t_1] \vdash_s e : t_2$. From the premise of the goal, we have $\mathsf{envtype}\ G\ E$. However, to show that $\vdash_s \mathsf{Clos}\ s\ e\ (E \lfloor_{\mathrm{fv}(e)} \setminus\setminus s) : \mathsf{fnT}\ t_1\ t_2$, we need to show that there exists a typing environment G' such that $\mathsf{envtype}\ G'\ (E \lfloor_{\mathrm{fv}(e)} \setminus\setminus s)$ holds (Definition 7). Clearly, the original G does not qualify, as its domain may be larger than that of $E \lfloor_{\mathrm{fv}(e)} \setminus\setminus s$. Thus, Corollary 2, which we will prove shortly, provides us with a minimal typing environment G' in which e has type *t2* and for which $\mathsf{envtype}\ G'\ (E \lfloor_{\mathrm{fv}(e)} \setminus\setminus s)$ holds, thereby proving the goal.

The first lemma states that if an expression e is typed in an environment G, then G contains type information for all free variables of e.
Lemma 7. - *typecheck_env_fv*

Proof: By rule induction on Definition 5. □

The second lemma states that if an expression e is typed in an environment *G1*, then e retains the same type in a smaller typing environment *G2*, provided that *G2* contains type information for all free variables in e.
Lemma 8. - *typecheck_env_submap*

Proof: By rule induction on Definition 5. □

The third lemma states that we can always type the expression e in a smaller typing environment, provided that the smaller typing environment still includes the bindings for the free variables of e.
Lemma 9. - *typecheck_drestrict*

Proof: Follows directly from Lemma 7 and Lemma 8. □

Lemma 9 and Lemma 7 provide the existence of a minimal typing environment for an expression e, containing exactly the bindings for all free variables of e.
Corollary 2. - *minimal typing env*

Proof: By Lemma 9, we know that if there exists some G such that $G \vdash_s e : t$, then we have $G \lfloor_{\mathrm{fv}(e)} \vdash_s e : t$. According to Lemma 7, G contains bindings for all free variables in e; thus, $G \lfloor_{\mathrm{fv}(e)}$ contains exactly the bindings for all free variables in e, given Definition 9.

For the reverse direction, suppose there exists a typing environment G' that contains fewer bindings than $G \lfloor_{\mathrm{fv}(e)}$, and we have $G' \vdash_s e : t$. By Lemma 7, G' must contain bindings for all free variables in e, leading to a contradiction. □

4.4.2 Type Soundness

We are now ready to prove the type soundness property for Sprinkles.

Theorem 1.

$$\begin{aligned}&\mathsf{envtype}\ G\ E\ \wedge\ G \vdash_s e : ty\ \Rightarrow\\ &(\exists v.\ \mathsf{eval\_exp}\ c\ E\ e = \mathsf{Value}\ v\ \wedge\ \vdash_s v : ty)\ \vee\\ &(\exists exn.\ \mathsf{eval\_exp}\ c\ E\ e = \mathsf{Exn}\ exn)\ \vee\\ &\mathsf{eval\_exp}\ c\ E\ e = \mathsf{Timeout}\end{aligned}$$

Proof: We prove this by induction on the definition of eval_exp. The case for Var follows directly from Definition 6. In the case of Let $s\ e_1\ e_2$, by Lemma 4, we have $G \vdash_s e_1 : t_0$ and $G[s := t_0] \vdash_s e_2 : t$, given that $G \vdash_s \mathsf{Let}\ s\ e_1\ e_2 : t$. Then, by the inductive hypothesis for *e1*, we know that *e1* evaluates to some value v and we have $\vdash_s v : t_0$. Lemma 6 then provides us with envtype $G[s := t_0]\ E[s := v]$. This allows us to apply the results from the inductive hypothesis to *e2*, thereby proving our goal.

For all following cases, we assume that Lemma 4 is applied first.

For Fn $s\ e$ case, given envtype $G\ E$ and $G[s := t_1] \vdash_s e : t_2$, we need to show that $\vdash_s \mathsf{Clos}\ s\ e\ (E \downharpoonright_{\mathrm{fv}(e)} \setminus\setminus s) : \mathsf{fnT}\ t_1\ t_2$. According to the definitions of value_type (Definition 7) and envtype (Definition 6), we must demonstrate the existence of a typing environment *G0* such that $G_0[s := t_1] \vdash_s e : t_2$ and envtype $G_0\ (E \downharpoonright_{\mathrm{fv}(e)} \setminus\setminus s)$.

By Corollary 2, we know there exists a minimal typing environment G' such that $G'[s := t_1] \vdash_s e : t_2$, and this environment contains bindings for all free variables of e, excluding the binding for s if $s \in \mathrm{fv}(e)$. Therefore, by the definition of DRESTRICT (Definition 9) and Definition 6, we have found such an environment *G0*, which is G'.

If we use an unrestricted function closure in Definition 8 instead, we need to show $\vdash_s \mathsf{Clos}\ s\ e\ E : \mathsf{fnT}\ t_1\ t_2$, which is straightforward given Definition 7.

For BinOp and Uop cases, results follow directly from Proposition 2 and Proposition 1, as discussed in Section 4.4.1.

For If $e\ e_1\ e_2$ case, the inductive hypothesis for the guard expression e tells us that e evaluates to a value v, where $\vdash_s v : \mathsf{boolT}$. By Lemma 5, we know that v must be either BoolV T or BoolV F. Both the true and false branches are covered by the inductive hypotheses.

For App $e_1\ e_2$ case, the inductive hypotheses for *e1* and *e2* tell us that they evaluate to *v1* and *v2*, respectively. We also know that $\vdash_s v_1 : \mathsf{fnT}\ t_1\ t_2$ and $\vdash_s v_2 : t_1$. By Lemma 5, we conclude that *v1* is a function closure, represented as Clos $s\ e\ E$, which matches the expressions in our goal. Additionally, Lemma 5 provides us with $G[s := t_1] \vdash_s e : t_2$, where envtype $G\ E$. Applying Lemma 6, we obtain envtype $G[s := t_1]\ E[s := v_2]$. This allows us to utilise the inductive hypothesis for e, which solves the goal.

For Case $e\ s_1\ e_1\ s_2\ e_2$ case, the inductive hypothesis tells us that e evaluates to a value v, where $\vdash_s v : \mathsf{sumT}\ t_1\ t_2$. By Lemma 5, we know that v is either SumLV v' or SumRV v' for some value v'.

In both scenarios, we have either $\vdash_s v' : t_1$ or $\vdash_s v' : t_2$. Consequently, by Lemma 6, we find either envtype $G[s_1 := t_1]\ E[s_1 := v']$ or envtype $G[s_2 := t_2]\ E[s_2 := v']$. This enables us to apply the inductive hypothesis for *e1* or *e2*, which solves the goal. □

Chapter 5

Strong Normalisation

We find that the strong normalization property of Sprinkles is essential for re-establishing the progress property in the enriched choreography. Without it, if local computation in one of the processes fails to terminate, the choreography will deadlock, waiting indefinitely for that process to complete.

The differences between our approach to proving strong normalisation and a more standard method involving simply typed lambda calculus with non-deterministic reductions, small-step semantics, and substitutions are discussed in Section ?. In brief, a straightforward induction over Definition 5 fails in the case of App e_1 e_2, because the e in the function closure Clos s e E of *e1* can be arbitrarily large. Instead, we use induction over the types of values, which leads us to define *strongly normalizing values*, as introduced in Definition 11.

Sprinkles's evaluation function is deterministic by using a functional big-step semantics, so the strong normalisation proof for Sprinkles is equivalent to the weak normalisation proof.

5.1 Formalisation

We now introduce the definitions of logical relations needed to state the Fundamental Lemma.

Strongly Normalising Values

In a standard simply typed lambda calculus, terms can be values, but in Sprinkles, values are not part of the expression syntax. Instead, in the recursive cases of our evaluation function (Definition 8), the values of sub-expressions are evaluated first and then used to evaluate the current expression. Therefore, we define sn_v t v as an inductive notion of strongly normalising values, indicating that these values demonstrate strong normalization behaviour of specific types during evaluation.

Definition 11. - *sn_v*

Literal values are always strongly normalising. For a closure value Clos s e E of type fnT t_1 t_2 to be strongly normalising, we require that if the expression e in $E[s := v]$ is evaluated, given that the value v of type *t1* is strongly normalising, then the evaluation of e must always return

a strongly normalising value of type *t2* or result in an exception. A pair value PairV v_1 v_2 of type pairT t_1 t_2 is strongly normalising if and only if both *v1* and *v2* are strongly normalising. A SumLV v value of type sumT t_1 t_2 is strongly normalising if and only if v is strongly normalising according to type *t1*. The condition for sn_v (sumT t_1 t_2) (SumRV v) is defined similarly.

Environments

Since we assign values to free variables by looking them up in an environment during expression evaluation, we introduce a constraint between the typing environment and the dynamic environment. This constraint requires the dynamic environment to store strongly normalising values that correspond to the variables and types specified by the typing environment.
Definition 12. - *envsn*

Evaluation of Expressions

We define the evaluation of an expression e of type t under the environment E as strongly normalising if and only if, for some initial clock c, the evaluation returns a strongly normalising value or an exception.
Definition 13.

$$\begin{aligned}&\mathsf{sn\_exec}\ t\ E\ e \stackrel{\text{def}}{=}\\ &(\exists\, cl\ v.\ \mathsf{eval\_exp}\ cl\ E\ e\ =\ \mathsf{Value}\ v\ \wedge\ \mathsf{sn\_v}\ t\ v)\ \vee\\ &\exists\, cl\ exn.\ \mathsf{eval\_exp}\ cl\ E\ e\ =\ \mathsf{Exn}\ exn\end{aligned}$$

We define the evaluation of an expression e, typed under the typing environment G with type t, as strongly normalising if and only if the evaluation of e is strongly normalising under any dynamic environment E that satisfies envsn $(G\ \lfloor_{\mathrm{fv}(e)})\ E$.
Definition 14. - *sn_e*

We use $G\ \lfloor_{\mathrm{fv}(e)}$ instead of G to accommodate the restricted environment used in closures, as defined in Definition 8. This adjustment is necessary to re-establish Proposition 3. Further discussion on Proposition 3 and the modifications made to restore it can be found in Section 6.6.1.

If we only require envsn $G\ E$ for any E in the definition of sn_e, the inductive hypothesis for the function expression Fn $s\ e$ becomes inapplicable. This prevents us from proving that the evaluated closure value is strongly normalising with respect to the type of the function expression, specifically sn_v (fnT $t\ t'$) (Clos $s\ e\ (E\ \lfloor_{\mathrm{fv}(e)}\ \backslash\backslash\ s)$). This is because, according to Definition 11, the body e in the function closure must be evaluated in the restricted dynamic environment $E\ \lfloor_{\mathrm{fv}(e)}\ \backslash\backslash\ s$ captured by the closure. Therefore, we need to instantiate the dynamic environment in the inductive hypothesis for e as $E\ \lfloor_{\mathrm{fv}(e)}\ \backslash\backslash\ s$. However, this necessitates first proving envsn $G[s := t]\ (E\ \lfloor_{\mathrm{fv}(e)}\ \backslash\backslash\ s)$ in order to apply the results of the inductive hypothesis for e. Unfortunately, envsn $G[s := t]\ (E\ \lfloor_{\mathrm{fv}(e)}\ \backslash\backslash\ s)$ cannot be deduced from envsn $G[s := t]\ E$, which is the premise for the current goal. This is because G may contain arbitrary variables that are not included in $E\ \lfloor_{\mathrm{fv}(e)}\ \backslash\backslash\ s$.

Thus, we modify the definition of sn_e to use a restricted typing environment that aligns with the restricted dynamic environment in a function closure. We require this restricted typing environment to include types for at most all free variables of the expression being evaluated. After introducing this restriction, the condition for the inductive hypothesis of e becomes envsn $(G\ \lfloor_{\mathrm{fv}(e)})[s := t]\ (E\ \lfloor_{\mathrm{fv}(e)})[s$ which can be deduced from envsn $(G\ \lfloor_{\mathrm{fv}(e)\setminus\{s\}})\ E$, the new premise of the current goal following the restriction of the typing environment.

5.2 Auxillary Lemmas

In this section, we present several lemmas related to the definitions introduced in Section 5.1. These lemmas will be utilised in the proof of the Fundamental Lemma.

Similar to the lemma of canonical forms for typed values (Lemma 5), we have a lemma of canonical forms for strongly normalising values. This lemma allows us to express a value v that satisfies sn_v t v for some t in a concrete syntax. This is particularly useful when proving Lemma 12 and 13. Additionally, Lemma 10 is used to prove the If, App, and Case cases in the Fundamental Lemma proof, as it enables us to represent the boolean guard value, function closure value, and sum value in their concrete syntax rather than simply referring to them as v.
Lemma 10. - *sn_v_**

Proof: Immediately follows from Definition 11. □

The following lemmas regarding envsn G E prove to be quite useful in the proof of the Fundamental Lemma, particularly when we need to pair G with a larger or smaller E while maintaining the envsn relation.
Lemma 11. *envsn lemmas*

1. *envsn_g_submap*
2. *envsn_update*
3. *envsn_e_submap2*
4. *envsn_e_submap*

Proof: Follows directly from Definition 12. □

Property 1 enables us to establish envsn G' E for some G' that is smaller than G. This is particularly useful when we have envsn $(G \restriction_{\text{fv}(e)})$ E and wish to deduce envsn $(G \restriction_{\text{fv}(e_1)})$ E and envsn $(G \restriction_{\text{fv}(e_2)})$ E in order to apply the inductive hypotheses for the sub-expressions *e1* and *e2* of e. This is because the free variables of an expression are typically defined as the union of the free variables of each of its sub-expressions. Property 2 is used in the Let and Sum cases of the Fundamental Lemma proof.

Property 3 allows us to establish the envsn property between G and a larger E. This lemma is used to re-stablish the envsn property required by the inductive hypotheses when the variable x, bound by Fn, Let, or Case, is not utilised in a later expression e. As a result, the variable x is not included in the typing environment for e, as x is not a free variable of e. However, our dynamic rules do not account for this information and still evaluate e with the previous environment, updated with the value of x. In this case, Property 1 is not directly applicable since we do not update the typing environment. Instead, with Property 3, we can extend the dynamic environment to include additional information about the values of unused variables.

Property 4 is used to address the restricted function closure in the Fn case of the Fundamental Lemma proof. The premise of our current goal provides us with envsn $(G \restriction_{\text{fv}(e)\setminus\{s\}})$ E, but we need to establish the envsn property for the restricted environment in the function closure Clos s e $(E \restriction_{\text{fv}(e)} \setminus\setminus s)$. So we can apply the inductive hypothesis for the function body expression e stored in the closure. Property 4 enables us to reduce the dynamic environment, provided that the modified environment still encompasses all variables included in the typing environment.

Finally, we require the last two lemmas to address the BinOp and Uop cases. In the BinOp *bop* e_1 e_2 case, if the inductive hypotheses indicate that *e1* and *e2* both evaluate to values *v1* and *v2*,

respectively, we need Lemma 12 to ensure that, given sn_v t_1 v_1 and sn_v t_2 v_2, the expression eval_bop *bop* v_1 v_2 always returns either a value or an exception. The same reasoning applies to the Uop case.

Lemma 12. - *bop_sn*

Proof: boptype is non-recursive. We prove by performing cases analyses based on boptype rules and each case follows from Lemma 10. □

Lemma 13. - *uop_sn*

Proof: Same as the above. □

5.3 The Fundamental Lemma

Using the definitions from Section 5.1, we state the Fundamental Lemma as:

Lemma 14. - *sn_lemma*

5.3.1 Strategy

We prove this using rule induction over Definition 5. The primary strategy in each case is to first derive the envsn property required by the inductive hypotheses from the premise of the current goal. Subsequently, we consider the sub-expressions of the current expression, which will either evaluate to a value or an exception based on the corresponding inductive hypotheses. If any sub-expression evaluates to an exception, the monadic binds employed in Definition 8 will ensure that the evaluation in the goal returns that exception.

We need to identify a clock value under which the first sub-expression that evaluates to an exception, along with all preceding sub-expressions that evaluate to values, continues to produce the same results. By applying Lemma 1 and 2, we can consistently use the summation of the clock values employed to evaluate these sub-expressions as the clock value for the evaluations in the goal statements, ensuring that they yield the same outcomes.

If no sub-expression evaluates to an exception, we either have an inductive hypothesis for the evaluation of some sub-expression that utilises the evaluated values, or we have an external eval_bop or eval_uop that makes use of those values. In the first case, we need to establish the envsn property required by the inductive hypothesis, which typically relies on Lemma 11. In the second case, we must apply Lemma 12 and 13.

The proof strategy itself is fairly straightforward; however, the evaluation of many expressions requires evaluating sub-expressions first (as outlined in Definition 8). This results in numerous similar cases, each requiring slightly different tactics, which contributes to the length of the HOL proof. For instance, the expression BinOp *bop* e_1 e_2 requires that we first evaluate *e1* and *e2* before calling eval_bop with the evaluated values. This leads to four distinct cases, as both *e1* and *e2* can evaluate to either a value or an exception.

Additionally, as discussed in Lemma 11 (3), if the bound variable x is not used in a sub-expression e, the envsn statement in the inductive hypothesis for e will be envsn $(G \restriction_{\text{fv}(e)})$ $E[x := v]$. In contrast, if x is used in e, the statement would be envsn $(G \restriction_{\text{fv}(e)})[x := t]$ $E[x := v]$. These two cases require slightly different proofs for reasons discussed in Section 5.2, and this added layer of distinction increases the number of generated cases, particularly for the Fn, Let, and Case scenarios.

5.3.2 Proof

Base cases. Evaluation for literals always return values and those values are related to the types of literal expressions by sn_v. For variable expressions, the goal follows from Definition 12.

Function. Consider a function expression Fn s e of type fnT t_1 t_2. Since we are using induction over type checking relation (Definition 5), we have one inductive hypothesis for evaluating e. According to Definition 8, a function expression always evaluates to a closure. We then need to show this closure is related to fnT t_1 t_2 by sn_v. According to Definition 11, we need to demonstrate that for any v such that sn_v t_1 v, evaluating e in the closure's environment extended with binding s to v either returns a value or an exception. This aligns with the result of the inductive hypothesis for e. From the discussion in Section 5.1, we know that the envsn property required to use the inductive hypothesis for e can be deduced from the premise of the current goal.

If expression. Consider an if expression If *bg* e_1 e_2. The If case follows the strategy outlined in Section 5.3.1. According to Definition 14, the condition for sn_exec t E (If *bg* e_1 e_2) requires envsn $(G \downarrow_{\text{fv}(\text{If } bg\ e_1\ e_2)})$ E. Since the free variables of If *bg* e_1 e_2 are the union of those in *bg*, *e1*, and *e2*, Lemma 11 (1) allows us to establish envsn $(G \downarrow_{\text{fv}(bg)})$ E, envsn $(G \downarrow_{\text{fv}(e_1)})$ E, and envsn $(G \downarrow_{\text{fv}(e_2)})$ E. With these, we can apply the inductive hypotheses for *bg*, *e1*, and *e2*.

Since each of them can evaluate to either a value or an exception, this generates 2^3 cases. If *bg* evaluates to an exception, then If *bg* e_1 e_2 will evaluate to an exception regardless of the outcomes of *e1* and *e2*. If *bg* evaluates to a value v, we apply the canonical form lemma (Lemma 10) to express v as BoolV b. Depending on the value of b and the evaluation results of *e1* and *e2* (which could be either a value or an exception as given by the corresponding inductive hypothesis), we use Lemma 1 and 2, instantiated with the clock value $c_{bg} + c_1 + c_2$, to prove the goal.

Binary operation. Consider a binary operation BinOp *bop* e_1 e_2. The BinOp case is similar to the If case and follows the strategy outlined in Section 5.3.1. From the premise of the current goal, we can deduce the necessary envsn conditions to use the inductive hypotheses for *e1* and *e2* by applying Lemma 11 (1). Since both *e1* and *e2* can evaluate to either a value or an exception, this generates 2^2 cases.

If either *e1* or *e2* evaluates to an exception, we use Lemma 1 and 2 with the clock value $c_1 + c_2$ to prove the goal. If both *e1* and *e2* evaluate to values *v1* and *v2* respectively, the inductive hypotheses provide sn_v t_1 v_1 and sn_v t_2 v_2. We then instantiate Lemma 1 with a clock value of $c_1 + c_2$ so that *e1* and *e2* are evaluated with the same clock value, reaching the same results *v1* and *v2*. Finally, Lemma 12 guarantees that eval_bop v_1 v_2 will return either a value or an exception, completing the proof.

Unary operation. Same as the binary operation case.

Let expression. Consider a Let expression Let s e_1 e_2. Similar to the previous cases, we use Lemma 11 (1) to deduce the envsn conditions required for applying the inductive hypothesis of *e1*. If *e1* evaluates to an exception, the proof is complete. If *e1* evaluates to a value v, we need to show that evaluating *e2* within $E[s := v]$ will either return a value or an exception. This outcome aligns with the inductive hypothesis for *e2*.

By applying Lemma 11 (2), we establish envsn $G[s := t_1]$ $E[s := v]$. This allows us to use the inductive hypothesis for *e2*, confirming that evaluating *e2* within $E[s := v]$ will result in either a value or an exception. Finally, by instantiating Lemma 1 and 2 with a clock value of $c_1 + c_2$, we complete the proof.

As discussed in Section 5.2 regarding Lemma 11 (3), if s is not used in *e2*, meaning s is not a free variable in *e2*, the inductive hypothesis for *e2* requires envsn G $E[s := v]$. Lemma 11 (3) allows us to deduce envsn G $E[s := v]$ in this situation.

Application. Application case follows the strategy stated in Section 5.3.1 and is similar to the If case.

Case expression. Consider a case expression Case e s_1 e_1 s_2 e_2. The proof follows the strategy outlined in Section 5.3.1, where Lemma 11 (1) is used to apply the inductive hypothesis for e. However, additional case analyses are needed to determine whether *s1* is a free variable in *e1* and whether *s2* is a free variable in *e2*. This additional analysis is similar to what is required in the Let case, where either Lemma 11 (3) or Lemma 11 (2) is used to apply the inductive hypothesis for each situation. □

5.3.3 Strong Normalisation

We now state the strong normalisation property for Sprinkles:

Corollary 3.

$$\begin{aligned}&\emptyset \vdash_s e : t \Rightarrow \\ &(\exists\, cl\ v\ E.\ \mathsf{eval\_exp}\ cl\ E\ e\ =\ \mathsf{Value}\ v\ \wedge\ \mathsf{sn\_v}\ t\ v)\ \vee \\ &\exists\, cl\ exn\ E.\ \mathsf{eval\_exp}\ cl\ E\ e\ =\ \mathsf{Exn}\ exn\end{aligned}$$

Proof: Follows from Lemma 14 □

Corollary 3 says that if a closed expression of Sprinkles is well-typed, then it either evaluates to a value or to an exception.

This result is nice on its own; however, Corollary 3 only establishes the existence of such an environment E. As a result, this finding is not directly applicable in the progress and type preservation proof for the enriched choreography. Instead, we aim to show that the evaluation of a typed expression terminates in the localised choreography environment. Theorem 3 in Section 6.4.2 provides this crucial result.

Chapter 6

Kalas Enriched with Sprinkles

We introduce the enrich Kalas, where local computations are handled by Sprinkles . As a result, the external function $f(v)$ in the previous Let rule of Kalas (Definition 1) is replaced with a Sprinkles expression e. Messages in Kalas used to be a list of strings, but now they are string values in Sprinkles. Since evaluations in Sprinkles may return an exception, we introduce new transition rules in the enriched Kalas to handle local computation failure.

We provide typing rules for the enrich Kalas. Our choreography does not return values and thus our rules only specify whether a choreography type checks. Kalas uses *states* to store values for variables located at a certain process, so we introduce a similar *typing state* that stores types for variables located at a certain process.

Since a local expression e neither cares neither about values of variables in other processes nor about their types, we introduce the concept of *localised* environment and *localised* typing environment to describe the result of filtering out values or types of variables in other processes. So e can be evaluated in the *localised* environment and typed in the *localised* typing environment.

We prove that our enriched Kalas has an important property: *progress*. The strong normalisation property of Sprinkles is crucial for the progress proof, otherwise we would have had to solve the halting problem. We also prove that the non-recursive, synchronous transitions in the enriched Kalas has type preservation property, where the type soundness and strong normalisation of local computations is required.

6.1 Syntax

Definition 15 (Syntax for Kalas enriched with Sprinkles)**.** Choreographies *in Kalas enriched with Sprinkles, ranged over by C, are inductively defined by the grammar*

C	$::=$	$p_1.v_1 \twoheadrightarrow p_2.v_2; C$	*(com)*	$p_1 \twoheadrightarrow p_2[b]; C$	*(sel)*
		if $v@p$ **then** C_1 **else** C_2	*(if)*	Let $v\ p\ e\ C$	*(let)*
		$\mu X.\ C$	*(fix)*	X	*(var)*
		$\mathbf{0}$	*(nil)*		

In Let $v\ p\ e\ C$, e represents a Sprinkles expression and (v, p) represents a variable v located at process p. The message values are Sprinkles's results (Definition 4.1). Other syntax are the same as Kalas' (Definition 1).

Free variables in a choreography are defined in Definition 16. Nil does not contain any free variables. Sel and Fix do not introduce any new free variables. In **if** $v@p$ **then** c_1 **else** c_2, the pair (v, p) becomes free. In $p_1.v_1 \twoheadrightarrow p_2.v_2; c$, the variable (v_1, p_1) is free, while (v_2, p_2) becomes bound. The free variables in Let $v\ p\ e\ c$ are defined as the union of all free variables in the expression e located at p and the free variables in the sequencing choreography c, excluding (v, p).

For Nil, IfThen, Sel, and Fix, the set of free variables in the current choreography is a superset of the free variables in the subsequent choreography. However, this does not necessarily hold for Com and Let. Therefore, as a closed choreography progresses, it may become open.
Definition 16 (Free variables of choreography)**.**

$$
\begin{aligned}
&\mathrm{fv}(\textbf{\textit{if}}\ v@p\ \textbf{\textit{then}}\ c_1\ \textbf{\textit{else}}\ c_2) \stackrel{\text{def}}{=} \{\ (v,p)\ \} \cup (\mathrm{fv}(c_1) \cup \mathrm{fv}(c_2))\\
&\mathrm{fv}(p_1.v_1 \twoheadrightarrow p_2.v_2; c) \stackrel{\text{def}}{=} \{\ (v_1,p_1)\ \} \cup (\mathrm{fv}(c) \setminus \{(v_2,p_2)\})\\
&\mathrm{fv}(\mathsf{Let}\ v\ p\ e\ c) \stackrel{\text{def}}{=} \{\ (s,p) \mid s \in \mathrm{fv}(e)\ \} \cup (\mathrm{fv}(c) \setminus \{(v,p)\})\\
&\mathrm{fv}(p \twoheadrightarrow q[b]; c) \stackrel{\text{def}}{=} \mathrm{fv}(c)\\
&\mathrm{fv}(\mu X.\ c) \stackrel{\text{def}}{=} \mathrm{fv}(c)
\end{aligned}
$$

6.2 Statics

6.2.1 Syntax and Typing Rules

A typing state, Γ, is a finite map that associates pairs (v, p) with types from Sprinkles's type system (see Definition 4). A type statement in the enriched Kalas is expressed as $\Gamma\ ,\ \Theta \vdash C\ \checkmark$, where Γ is the typing state, Θ is a set of processes, and C is the choreography being typed.

We also introduce the *localised* typing environment, localise $\Gamma\ p$, for typing Sprinkles expressions. Since a Sprinkles expression e, located at a process p, only depends on variables at p, we filter out variables associated with other processes and retain only the types of variables located at p in localise $\Gamma\ p$. This procedure is formally defined in Definition 17.
Definition 17 (Localised environments)**.** *Given a state or a typing state s and a process p, we define the localise environment as a function composition*

$$
\mathsf{localise}\ s\ p \stackrel{\text{def}}{=} s \circ (\lambda\, vn.\ (vn, p))
$$

Definition 18 (Typing rules for Kalas enriched with Sprinkles)**.**

$$\text{CT-Com}\ \frac{\Gamma\ (v_1, p_1)\ =\ \mathsf{strT} \quad \{\ p_1;\ p_2\ \}\ \subseteq\ \Theta \quad p_1\ \neq\ p_2 \quad \Gamma[(v_2, p_2)\ :=\ \mathsf{strT}],\ \Theta\ \vdash\ c\ \checkmark}{\Gamma\ ,\ \Theta\ \vdash\ p_1.v_1\ \twoheadrightarrow\ p_2.v_2; c\ \checkmark}$$

$$\text{CT-If}\ \frac{\Gamma\ (v, p)\ =\ \mathsf{boolT} \quad \Gamma\ ,\ \Theta\ \vdash\ c_1\ \checkmark \quad \Gamma\ ,\ \Theta\ \vdash\ c_2\ \checkmark}{\Gamma\ ,\ \Theta\ \vdash\ \textbf{if}\ v@p\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \checkmark}$$

$$\text{CT-Let}\ \frac{\mathsf{localise}\ \Gamma\ p\ \vdash_s\ e\ :\ ety \quad \Gamma[(v, p)\ :=\ ety],\ \Theta\ \vdash\ c\ \checkmark}{\Gamma\ ,\ \Theta\ \vdash\ \mathsf{Let}\ v\ p\ e\ c\ \checkmark}$$

$$\text{CT-Sel}\ \frac{\{\ p_1;\ p_2\ \}\ \subseteq\ \Theta \quad p_1\ \neq\ p_2 \quad \Gamma\ ,\ \Theta\ \vdash\ c\ \checkmark}{\Gamma\ ,\ \Theta\ \vdash\ p_1\ \twoheadrightarrow\ p_2[b]; c\ \checkmark} \qquad \text{CT-Nil}\ \frac{}{\Gamma\ ,\ \Theta\ \vdash\ \mathbf{0}\ \checkmark}$$

$$\text{CT-Fix}\ \frac{\Gamma\ ,\ \Theta\ \cup\ \{\ dn\ \}\ \vdash\ c\ \checkmark}{\Gamma\ ,\ \Theta\ \vdash\ \mu dn.\ c\ \checkmark} \qquad \text{CT-Call}\ \frac{\{\ dn\ \}\ \subseteq\ \Theta}{\Gamma\ ,\ \Theta\ \vdash\ dn\checkmark}$$

The typing rules for the enriched Kalas are presented in Definition 18. **0** always type-checks, regardless of the context provided by Γ and Θ. For $p_1.v_1 \twoheadrightarrow p_2.v_2; c$, the sender *p1*'s variable *v1* must have the type strT, and the subsequent choreography c must be type-checked under the condition that the receiver *p2*'s variable *v2* holds a value of type strT. Note that self-communication is not permitted. In the case of Let, the Sprinkles expression e must be typed within a localised typing environment at the process p. The result of the local computation from e will be stored in the variable v at p. Therefore, the choreography c needs to type-check given the type of v at p.

Since local computations in the enriched Kalas assign values to free variables by looking them up in the localised environment, we define chorEnvtype Γ s to ensure that if a choreography is typed by Γ, then for any process p, we have envtype (localise Γ p) (localise s p). Combined with the Let typing rule in Definition 18, this guarantees type soundness for local computations. Additionally, chorEnvtype Γ s ensures that the state s contains values for all free variables in the choreography typed under Γ.

Definition 19 (Typing constrains for localised environments)**.** *Given a typing state Γ and a state s,* chorEnvtype Γ s *is defined as*

$$\mathsf{chorEnvtype}\ \Gamma\ s\ \stackrel{\mathsf{def}}{=}\ \forall\, p.\ \mathsf{envtype}\ (\mathsf{localise}\ \Gamma\ p)\ (\mathsf{localise}\ s\ p)$$

6.2.2 Typing Examples

The choreography in Example 1.0.2 can be typed using the rules from Definition 18, given an initial typing state of $\Gamma = (var, server) : \mathsf{strT}$ and a set of processes $\Theta = server, client$.

Example 6.2.1. - type the choreography in example 1

If *server* stores a non-string value in *var*, the choreography cannot be typed because the condition $\Gamma\ (var, server)\ =\ \mathsf{strT}$ would not be satisfied.

6.3 Dynamics

Our transition rules are similar to those of Kalas, with a few modifications. The CS-Com rule requires that (v_1, p_1) holds a Sprinkles value of type StrV. If this condition is not met, a communication exception is triggered, and the choreography transitions to Nil.

According to the CS-Let rule, if the expression e at process p evaluates to a value v within the localised environment at p, and if the localised environment contains bindings for all free variables in e, then the state s updates to store the value v in variable v at p, and the choreography proceeds to c. The additional condition, $\mathrm{fv}(e) \subseteq \texttt{domain}(\mathsf{localise}\ s\ p)$, is necessary to maintain Proposition 3, as discussed in Section 6.6. The outcome of the local evaluation is recorded in the transition label.

The C-Letexn rule is similar to C-Let; if a local evaluation fails, the choreography transitions to Nil. In C-Iftrue and C-Iffalse, we rely on Sprinkles's boolean values, and the rules are straightforward. If the value of variable v at p is not a boolean, a choreography failure occurs, causing a transition to Nil.

Definition 20 (Transition rules for Kalas enriched with Sprinkles)**.**

$$\text{CS-Com}\ \frac{s\ (v_1, p_1)\ =\ \mathsf{StrV}\ d \quad p_1 \neq p_2}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; c \xrightarrow[\epsilon]{p_1.v_1 \gg p_2.v_2} s[(v_2, p_2)\ :=\ \mathsf{StrV}\ d] \rhd c}$$

$$\text{CS-ComExn}\ \frac{s\ (v_1, p_1)\ =\ v \quad \forall s.\ v \neq \mathsf{StrV}\ s}{s \rhd p_1.v_1 \twoheadrightarrow p_2.v_2; c \xrightarrow[\epsilon]{\mathsf{LComExn}\ p_1\ v_1\ p_2\ v_2} s \rhd \mathbf{0}}$$

$$\text{CS-Let}\ \frac{\mathsf{eval\_exp}\ cl\ (\mathsf{localise}\ s\ p)\ e\ =\ \mathsf{Value}\ ev \quad \mathrm{fv}(e) \subseteq \texttt{domain}(\mathsf{localise}\ s\ p)}{s \rhd \mathsf{Let}\ v\ p\ e\ c \xrightarrow[\epsilon]{\texttt{let}\ v@p,(\mathsf{Value}\ ev)} s[(v, p)\ :=\ ev] \rhd c}$$

$$\text{CS-LetExn}\ \frac{\mathsf{eval\_exp}\ cl\ (\mathsf{localise}\ s\ p)\ e\ =\ \mathsf{Exn}\ exn \quad \mathrm{fv}(e) \subseteq \texttt{domain}(\mathsf{localise}\ s\ p)}{s \rhd \mathsf{Let}\ v\ p\ e\ c \xrightarrow[\epsilon]{\texttt{let}\ v@p,(\mathsf{Exn}\ exn)} s \rhd \mathbf{0}}$$

$$\text{CS-IfTrue}\ \frac{s\ (v, p)\ =\ \mathsf{BoolV\ T}}{s \rhd \textbf{if}\ v@p\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow[\epsilon]{\tau_p} s \rhd c_1}$$

$$\text{CS-IfFalse}\ \frac{s\ (v, p)\ =\ \mathsf{BoolV\ F}}{s \rhd \textbf{if}\ v@p\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow[\epsilon]{\tau_p} s \rhd c_2}$$

$$\text{CS-IfExn}\ \frac{s\ (v, p)\ =\ w \quad \neg\mathsf{is\_BoolV}\ w}{s \rhd \textbf{if}\ v@p\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \xrightarrow[\epsilon]{\tau_p} s \rhd \mathbf{0}}$$

6.4 Progress

In this section, we demonstrate that a typed choreography can always make progress, proving that the enriched Kalas with Sprinkles is deadlock-free. However, since a closed choreography can transition to an open one, we formalise our progress statement to cover general choreographies.

To ensure that the state s includes values for all free variables in a typed choreography, we use $\mathsf{chorEnvtype}\ \Gamma\ s$. The concept $\mathsf{chorEnvsn}\ \Gamma\ s$ is defined similarly to chorEnvtype, but it ensures that

s contains values that are strongly normalising, adhering to the typing constraints specified in Γ. These constraints play a crucial role in proving the progress theorem.

We also establish that the evaluation of typed Sprinkles expressions within a localised environment always results in either a value or an exception. This outcome is guaranteed by the strong normalisation property of Sprinkles, which is essential for the progress proof.

6.4.1 Formalisation

First, we define $\mathsf{chorEnvsn}\ \Gamma\ s$, which stipulates that the state s must contain strongly normalising values for all free variables in the choreography typed under Γ.
Definition 21 (Strong normalisation constrains for localised environments)**.** *Given a typing state Γ and a state s,* $\mathsf{chorEnvsn}\ \Gamma\ s$ *is defined as*

$$\mathsf{chorEnvsn}\ \Gamma\ s \stackrel{\text{def}}{=} \forall\, p.\ \mathsf{envsn}\ (\mathsf{localise}\ \Gamma\ p)\ (\mathsf{localise}\ s\ p)$$

We state the progress property for Kalas enriched with Sprinkles as the following:
Theorem 2 (Progress)**.** *Given a typing state Γ, a set of processes Θ, and an initial state s, we state the progress property as*

$$\begin{array}{l}\Gamma\ ,\ \Theta\ \vdash\ c\ \checkmark\ \wedge\ \mathsf{chorEnvtype}\ \Gamma\ s\ \wedge\ \mathsf{chorEnvsn}\ \Gamma\ s\ \Rightarrow \\ \quad \exists\, \tau\ l\ s'\ c'.\ s \rhd c\ \xrightarrow[l]{\tau}\ s' \rhd c'\ \vee\ \neg(c\ \neq\ \mathbf{0}\ \wedge\ \forall\, x.\ c\ \neq\ x)\end{array}$$

It states that for a choreography C typed under the typing state Γ, if the corresponding state s contains variables typed in Γ with values that are both of the correct types and strongly normalising, then C can either advance from state s or be equal to Nil.

One might expect that any closed choreography that can be typed will always make progress, in line with the progress theorem. However, we know that a closed choreography may transition into an open choreography, as discussed in Section 6.1. This nuance diminishes the interest of such a statement. Furthermore, according to the transition rules of the enriched Kalas (Definition 20), the state s may be extended after a transition, meaning that a closed choreography starting from an empty state could end up in a non-empty state. Consequently, we frame the progress theorem for more general choreographies, which requires the addition of the constraints chorEnvtype and chorEnvsn.

6.4.2 Lemmas

We introduce several lemmas that are useful for the progress proof. The first lemma is the generation lemma, which enables us to derive types and typing information for variables, processes, and choreographies from a typed choreography.
Lemma 15 (Generation lemma)**.** *Given a typed choreography, we have*

1. $$\begin{array}{l}\Gamma\ ,\ \Theta\ \vdash\ \mathsf{Let}\ v\ p\ e\ c\ \checkmark\ \iff \\ \exists\, ety.\ \mathsf{localise}\ \Gamma\ p\ \vdash_s\ e\ :\ ety\ \wedge\ \Gamma[(v,p)\ :=\ ety],\ \Theta\ \vdash\ c\ \checkmark\end{array}$$

2. $$\begin{array}{l}\Gamma\ ,\ \Theta\ \vdash\ \textbf{\textit{if}}\ v@p\ \textbf{\textit{then}}\ c_1\ \textbf{\textit{else}}\ c_2\ \checkmark\ \iff \\ \Gamma\ (v,p)\ =\ \mathsf{boolT}\ \wedge\ \Gamma\ ,\ \Theta\ \vdash\ c_1\ \checkmark\ \wedge\ \Gamma\ ,\ \Theta\ \vdash\ c_2\ \checkmark\end{array}$$

3.
$$\Gamma\ ,\ \Theta\ \vdash p_1.v_1 \twoheadrightarrow p_2.v_2; c\ \checkmark \iff \Gamma\ (v_1, p_1)\ =\ \mathsf{strT}\ \wedge\ \{\ p_1;\ p_2\ \}\ \subseteq\ \Theta\ \wedge\ p_1\ \neq\ p_2\ \wedge\ \Gamma[(v_2, p_2)\ :=\ \mathsf{strT}],\ \Theta\ \vdash\ c\ \checkmark$$

4.
$$\Gamma\ ,\ \Theta\ \vdash p_1 \twoheadrightarrow p_2[b]; c\ \checkmark \iff \{\ p_1;\ p_2\ \}\ \subseteq\ \Theta\ \wedge\ p_1\ \neq\ p_2\ \wedge\ \Gamma\ ,\ \Theta\ \vdash\ c\ \checkmark$$

5.
$$\Gamma\ ,\ \Theta\ \vdash \mu dn.\ c\ \checkmark \iff \Gamma\ ,\ \Theta\ \cup\ \{\ dn\ \}\ \vdash\ c\ \checkmark$$

Proof: Immediately follows from Definition 18. □

The Let $v\ p\ e\ c$ case of the generation lemma is required by the Let case in the progress proof. Lemma 15 gives us that the local expression e is typed if the choreography is typed. This allows us to apply the strong normalisation theorem to e. Lemma 15 is also used in the type preservation proof.

The following lemma states that if a choreography C is typed under some Γ, then it does not exhibit any self-communication behaviours. Furthermore, if we have chorEnvtype $\Gamma\ s$, then the state s contains correctly typed values for all free variables in C.

Lemma 16 (Well-formedness)**.** *Given a typing state Γ, a set of processes Θ, a state s, we have the following properties for a typed choreography c*

1. $\Gamma\ ,\ \Theta \vdash c\ \checkmark \Rightarrow$ no_self_comunication c
2. $\Gamma\ ,\ \Theta \vdash c\ \checkmark \wedge$ chorEnvtype $\Gamma\ s \Rightarrow \mathrm{fv}(c) \subseteq domain(s)$

Proof: By rule induction over Definition 18. □

Strong Normalisation in the Localised Environments

As discussed in Section 5.3.3, Corollary 3 shows the strong normalization property of the Sprinkles evaluation function. However, it cannot be applied in the progress or type preservation proofs because we need to evaluate expressions in the localised environment, and Corollary 3 does not guarantee that evaluation in a localised environment will terminate. Therefore, we require Theorem 3, which states that if an expression e is typed in a typing environment localised at p, and if we have chorEnvsn $\Gamma\ s$, then evaluating e in the dynamic environment localised at p will either return a value or raise an exception.

Theorem 3 (Strong normalisation in localised environmet)**.**

$$\mathsf{chorEnvsn}\ \Gamma\ s\ \wedge\ \mathsf{localise}\ \Gamma\ p\ \vdash_s\ e\ :\ t\ \Rightarrow (\exists\ cl\ v.\ \mathsf{eval\_exp}\ cl\ (\mathsf{localise}\ s\ p)\ e\ =\ \mathsf{Value}\ v\ \wedge\ \mathsf{sn\_v}\ t\ v)\ \vee \exists\ cl\ exn.\ \mathsf{eval\_exp}\ cl\ (\mathsf{localise}\ s\ p)\ e\ =\ \mathsf{Exn}\ exn$$

Proof: Follows from the Fundamental Lemma (Lemma 14). □

6.4.3 Proof

We prove Theorem 2 through case analysis of the choreography syntax. From $\Gamma\ ,\ \Theta\ \vdash\ C\ \checkmark$ and chorEnvtype $\Gamma\ s$, we know that C does not contain any self-communications, and that state

s contains correctly typed values for all free variables in C, as established in Lemma 16. The cases for Sel and Fix follow directly from this result.

In the case of **if** $v@p$ **then** c_1 **else** c_2, we know that s stores a value for the variable v at process p. This value can take one of three forms: BoolV T, BoolV F, or any other type, corresponding to CS-IFTRUE, CS-IFFALSE, and CS-IFEXN, respectively. A similar analysis applies to the case of Com.

In the case of Let v p e c, Lemma 15 tells us that e is typed in localise Γ p. This allows us to apply Theorem 3, which states that e will either evaluate to a value or produce an exception. As discussed in Section 6.3, to apply SC-LET or SC-LETEXN, we must show that $\text{fv}(e) \subseteq \texttt{domain}(\text{localise } s\ p)$. Lemma 7 indicates that the localised typing environment localise Γ p contains types for the free variables of e. Given chorEnvtype Γ s and Definition 19, we know that the domain of Γ is a subset of the domain of s. Therefore, SC-LET or SC-LETEXN can be applied, solving the goal. □

6.5 Type Preservation

We formalise the type preservation theorem as:

Theorem 4 (Type preservation)**.** *Given a typing state Γ, a set of processes Θ, and an initial state s,*

$$\begin{array}{l}(c \neq \mathbf{0} \wedge \forall x.\ c \neq x) \wedge \text{chorEnvsn } \Gamma\ s \wedge \text{chorEnvtype } \Gamma\ s \wedge \\ \Gamma\ ,\ \Theta \vdash c\ \checkmark \wedge s \rhd c \xrightarrow[l]{\tau} s' \rhd c' \Rightarrow \\ \quad \exists \Gamma'. \\ \qquad \text{chorEnvsn } \Gamma'\ s' \wedge \text{chorEnvtype } \Gamma'\ s' \wedge \Gamma'\ ,\ \Theta \vdash c'\ \checkmark\end{array}$$

Theorem 4 states that if a choreography is typed under Γ and state s contains strongly normalising values of the correct types according to Γ, then when the choreography transitions to the next state, there exists a typing state Γ' such that the new choreography is typed and the new state s' contains strongly normalising values of the correct types according to Γ'.

We cannot guarantee that the new choreography will still be typed in the same typing state, as advancing the choreography may introduce new free variables. This is the same reason we formulated the progress theorem for a more general choreography rather than a closed one, as discussed in Section 6.4.1.

Proof: We prove Theorem 4 by induction on the transition relation defined in Definition 20. The cases for Nil, Sel, and If follow directly from Definitions 18 and 20.

To prove the cases for Com and Let, we need the following lemma about chorEnvtype and chorEnvsn:

Lemma 17 (Updating states)**.** *Given a typing state Γ, a state s, and some value v of type t,*

1. chorEnvtype Γ $s \wedge \vdash_s v : t \Rightarrow$
   $\forall\, vn\ p.$ chorEnvtype $\Gamma[(vn,p) := t]\ s[(vn,p) := v]$
2. chorEnvsn Γ $s \wedge$ sn_v t $v \Rightarrow$
   $\forall\, vn\ p.$ chorEnvsn $\Gamma[(vn,p) := t]\ s[(vn,p) := v]$

Proof: Follows from Definition 19 and 21. □

In the case of $p_1.v_1 \twoheadrightarrow p_2.v_2; c$, we know that CS-COM must apply since the choreography type checks, indicating that the variable $v1$ at p maps to a string value StrV d. By Lemma 15, we

have $\Gamma[(v_2, p_2) := \mathsf{strT}]$, $\Theta \vdash c \checkmark$. Furthermore, by Lemma 17, we obtain

$$\mathsf{chorEnvtype}\ \Gamma[(v_2, p_2) := \mathsf{strT}]\ s[(v_2, p_2) := \mathsf{StrV}\ d]$$

and $\mathsf{chorEnvsn}\ \Gamma[(v_2, p_2) := \mathsf{strT}]\ s[(v_2, p_2) := \mathsf{StrV}\ d]$, which match the goal.

In the case of $\mathsf{Let}\ x\ p\ e\ c$, if e evaluates to an exception, the proof is straightforward, as we transition into Nil when an exception occurs, and Nil can be typed by any Γ and Θ. When e evaluates to a value v, Lemma 15 tells us that $\mathsf{localise}\ \Gamma\ p \vdash_s e : t$ holds, along with $\Gamma[(x, p) := t]$, $\Theta \vdash c \checkmark$. If we can show that $\vdash_s v : t$ and $\mathsf{sn\_v}\ t\ v$, then applying Lemma 17 will complete the proof.

Since e is typed in the localised typing environment $\mathsf{localise}\ \Gamma\ p$, Theorem 3 and the type soundness of Sprinkles (Theorem 1) guarantee that e evaluates to either a value v' or an exception. In the case of an exception, as discussed above, the choreography transitions to Nil, making the proof straightforward. For the former scenario, we have $\vdash_s v' : t$ and $\mathsf{sn\_v}\ t\ v'$. Next, we need to show that $v = v'$, which holds true since the evaluation function for Sprinkles is deterministic.

We provide proof sketches for the parallel transitions (rules can be found in Appendix ?) even though we did not implement the proofs for parallel transitions in HOL4.

For the case of If, consider a choreography **if** $v@p$ **then** c_1 **else** c_2. We know the choreography type checks, and by Lemma 15, we have $\Gamma\ (v, p) = \mathsf{boolT}$, Γ , $\Theta \vdash c_1 \checkmark$, and Γ , $\Theta \vdash c_2 \checkmark$.

By the inductive hypotheses for *c1* and *c2*, we obtain Γ_1 , $\Theta \vdash c_1' \checkmark$ and Γ_2 , $\Theta \vdash c_2' \checkmark$. Additionally, we have $\mathsf{chorEnvsn}\ \Gamma_1\ s'$, $\mathsf{chorEnvsn}\ \Gamma_2\ s'$, $\mathsf{chorEnvtype}\ \Gamma_1\ s'$, and $\mathsf{chorEnvtype}\ \Gamma_2\ s'$.

By Definitions 21 and 19, we know that *Γ1* and *Γ2* assign the same type to variables in the intersection of their domains. Therefore, we can construct a Γ' that includes all mappings from *Γ1* and *Γ2* without encountering any contradictions. Consequently, we have $\mathsf{chorEnvsn}\ \Gamma'\ s'$ and $\mathsf{chorEnvtype}\ \Gamma'\ s'$.

By If-swap, we can ensure that the parallel transitions at *c1* and *c2* do not involve the variable v at p. Thus, $(s\ (v, p)) = (s'\ (v, p))$. As a result, **if** $v@p$ **then** c_1' **else** c_2' can be typed by $\Gamma'[(v, p) := \mathsf{boolT}]$. Similar analyses apply to the other parallel transition rules. □

However, we do not provide a proof for type preservation in the case of Fix, as our typing system cannot type an unfolding choreography. Additionally, we do not address the transitions involving asynchronous behaviours due to time constraints in this project.

6.6 Discussions on Kalas Properties

6.6.1 Transitions in Larger States

The Kalas repository reveals an interesting property: If a larger state *z2* contains *z1*, the choreography c will still transition into c', with z' containing *z1′*.

Proposition 3 (Transitions in a larger state)**.** *Given a state z1 and a larger state z2, we have*

$$z_1 \rhd c \xrightarrow[\tau]{\alpha} z_1' \rhd c' \ \wedge\ z_1 \sqsubseteq z_2 \Rightarrow \exists\, z'.\ z_2 \rhd c \xrightarrow[\tau]{\alpha} z' \rhd c' \ \wedge\ z_1' \sqsubseteq z'$$

Re-establishing this property requires two modifications.

Restricted Function Closure

The first modification involves changing the evaluation rule for function expressions in Sprinkles' (Definition 8) as mentioned in Section 4.3.1. This adjustment stipulates that a restricted environment, Clos $s\ e\ (E \downarrow_{\mathrm{fv}(e)} \setminus\setminus s)$, is stored within the evaluated closure. This change is crucial because evaluating Fn $s\ E$ into Clos $s\ e\ E$ could invalidate Proposition 3. A larger state might yield a closure value that contains a larger environment, preventing the choreography from transitioning to the same c' when given z'.

This explains why it is beneficial to restrict E to include only bindings for the free variables of e, as the set of free variables in e remains unchanged, even when we are given a larger state.

We also argue that it is essential to exclude s from the environment. If we only use Clos $s\ e\ (E \downarrow_{\mathrm{fv}(e)})$, the state z may lack a binding for s when evaluating Fn $s\ E$. Consequently, localise $z\ p \downarrow_{\mathrm{fv}(e)}$ may not contain a binding for s when $s \in \mathrm{fv}(e)$. In contrast, localise $z'\ p \downarrow_{\mathrm{fv}(e)}$ may contain a binding for s, as z' is larger. This means that localise $z'\ p \downarrow_{\mathrm{fv}(e)}$ may be larger than localise $z\ p \downarrow_{\mathrm{fv}(e)}$.

Another potential approach to re-establish Proposition 3 is to define an equivalence relation on function closures. In this framework, closures with the same parameter and function body are considered equivalent if their environments contain identical bindings for the free variables in the function body, while excluding any bindings for the function parameter itself. However, this method introduces additional complexity, as it requires addressing choreographies that contain different yet equivalent function closures.

Using a restricted function closure requires employing a minimal typing environment (Corollary 2) in the Fn case to ensure type soundness (as discussed in Section 4.4.2). Additionally, this restricted closure environment requires the use of a limited typing environment, $G \downarrow_{\mathrm{fv}(e)}$, in the definition of sn_e (Definition 14). If we opt for Clos $s\ e\ E$ instead, we can forgo both Corollary 2 and the modifications to Definition 14. However, this alternative would invalidate Proposition 3 by the reasons discussed above.

Modified Transition Rules for the Enriched Kalas

We introduce an additional constraint, $\mathrm{fv}(e) \subseteq$ `domain`(localise $z\ p$), to the CS-Let and CS-Letexn rules for the enriched Kalas (as defined in Definition 20 in Section 6.3) to re-establish Proposition 3. In the case of CS-Let, we can evaluate a function expression Fn $s\ e$ into a closure value, regardless of the localised environment provided. This means that localise $z'\ p \downarrow_{\mathrm{fv}(e)} \setminus\setminus s$ can still be larger than localise $z\ p \downarrow_{\mathrm{fv}(e)} \setminus\setminus s$. Specifically, when localise $z\ p$ does not include all the free variables of e, localise $z'\ p$ may contain additional bindings for these free variables that are absent in localise $z\ p$.

However, this modification does not require any extra results to prove the Let transitions in the progress and type soundness proofs. This indicates that if we incorporate typing into the statement of Proposition 3, the changes made to the C-Let and C-Letexn rules can be omitted.

6.6.2 Synchronous Transitions

We have fixed the proof for Proposition 4 in Kalas' repository, which may be important for Endpoint Projection. The proposition states that if the state s contains values for all free variables of c, and c does not involve any self-communications, then for any label τ, the choreography can always advance synchronously until that label τ is consumed by a transition. If this does not occur, the choreography will advance to an end state, represented as $\mathbf{0}$.

Definition 22 (syncTrm)**.**

1.

$$
\begin{array}{l}
\mathsf{syncTrm}\ k\ (s, p_1.v_1 \mathrel{->} p_2.v_2; c)\ \tau \stackrel{\text{def}}{=} \\
\quad \mathit{if}\ k\ =\ 0\ \mathit{then}\ \mathsf{None} \\
\quad \mathit{else\ if}\ \mathsf{chor\_match}\ \tau\ (p_1.v_1 \mathrel{->} p_2.v_2; c)\ \mathit{then} \\
\quad\ \mathsf{Some}\ (\mathsf{chor\_tl}\ s\ (p_1.v_1 \mathrel{->} p_2.v_2; c)) \\
\quad \mathit{else} \\
\quad\ \mathit{case}\ \mathsf{some}\ \mathit{str}.\ s\ (v_1, p_1)\ =\ \mathsf{StrV}\ \mathit{str}\ \mathit{of} \\
\quad\quad \mathsf{None}\ \Rightarrow\ \mathsf{None} \\
\quad\quad \mid\ \mathsf{Some}\ \mathit{str}\ \Rightarrow\ \mathsf{syncTrm}\ (k\ -\ 1)\ (s[(v_2, p_2)\ :=\ \mathsf{StrV}\ \mathit{str}], c)\ \tau
\end{array}
$$

2.

$$
\begin{array}{l}
\mathsf{syncTrm}\ k\ (s, \mathsf{Let}\ v\ p\ e\ c)\ \tau \stackrel{\text{def}}{=} \\
\quad \mathit{if}\ k\ =\ 0\ \vee\ \mathsf{is\_bad\_label}\ \tau\ \mathit{then}\ \mathsf{None} \\
\quad \mathit{else\ if}\ \mathsf{chor\_match}\ \tau\ (\mathsf{Let}\ v\ p\ e\ c)\ \mathit{then} \\
\quad\ \mathit{case} \\
\quad\quad \mathsf{some}\ r.\ \exists\, cl. \\
\quad\quad\quad \mathsf{eval\_exp}\ cl\ (\mathsf{localise}\ s\ p)\ e\ =\ r\ \wedge\ r\ \neq\ \mathsf{Timeout} \\
\quad\ \mathit{of} \\
\quad\quad \mathsf{None}\ \Rightarrow\ \mathsf{None} \\
\quad\quad \mid\ \mathsf{Some}\ (\mathsf{Value}\ ev)\ \Rightarrow\ \mathsf{Some}\ (s[(v, p)\ :=\ ev], c) \\
\quad\quad \mid\ \mathsf{Some\ TypeError}\ \Rightarrow\ \mathsf{None} \\
\quad\quad \mid\ \mathsf{Some}\ (\mathsf{Exn}\ v_6)\ \Rightarrow\ \mathsf{Some}\ (s, \mathbf{0}) \\
\quad\quad \mid\ \mathsf{Some\ Timeout}\ \Rightarrow\ \mathsf{None} \\
\quad \mathit{else}\ \mathsf{syncTrm}\ (k\ -\ 1)\ (\mathsf{chor\_tl}\ s\ (\mathsf{Let}\ v\ p\ e\ c))\ \tau
\end{array}
$$

Proposition 4 (Synchronous transitions up to a given label)**.** *Given an initial state s, a choreography c, an integer k, and a label τ, we have*

$$
\begin{array}{l}
\mathrm{fv}(c)\ \subseteq\ \mathit{domain}(s)\ \wedge\ \mathsf{no\_self\_comunication}\ c\ \wedge \\
\mathsf{syncTrm}\ k\ (s, c)\ \tau\ =\ \mathsf{Some}\ (s', c')\ \Rightarrow\ \mathsf{trans\_sync}\ (s, c)\ (s', c')
\end{array}
$$

The function syncTrm takes a counter k, a state-choreography pair (s, c), and a label τ. It advances the choreography synchronously until the label τ is consumed, returning the choreography after this consumption. If the choreography reaches its end or if the counter decreases to zero, it returns None instead.

We modify the cases for If, Com, and Let within syncTrm to align with the transition rules for the enriched Kalas (as defined in Definition 20). The function chor_match τ c checks whether τ matches c; if it does, a synchronous transition from c will consume the label τ. The function chor_tl s c takes a state and a choreography, returning the updated state-choreography pair after a synchronous transition from (s, c).

Additionally, is_bad_label τ checks if τ is equal to `let` $v@p$, Timeout, which should not occur given the strong normalisation property of Sprinkles. Lastly, trans_sync (s, c) (s', c') is defined as the reflective transitive closure on synchronous transitions (i.e., transitions where the list of asynchronous labels is empty; see also Definition 20 and the rules for parallel transitions in Appendix ?). Definitions for chor_match and chor_tl can be found in Appendix ?.

The proof for Proposition 4 follows from the definitions above. For the Let case, is_bad_label τ helps us filter out `let` $v@p$, Timeout, as by Definition 22

$$\mathsf{syncTrm}\ k\ (s, \mathsf{Let}\ v\ p\ e\ c)\ (\texttt{let}\ v@p, \mathsf{Timeout})$$

will return None. □

Chapter 7

Concluding Remarks

7.1 Conclusion

In this thesis, we present a simply typed lambda calculus, Sprinkles, with functional big-step semantics and function closures. To restore the property for transitions in the larger state (Proposition **??**) of Kalas, we adjust the evaluation rule in Sprinkles. Specifically, when evaluating a function expression, we return a restricted function closure.

We establish that Sprinkles enjoys both type soundness and strong normalisation. Our type soundness proof follows a standard approach. However, evaluating function expressions into restricted closures requires us to exhibit a minimal typing environment for a typed expression in the function case of the type soundness proof.

Function big-step semantics means that the evaluation function for Sprinkles is total and deterministic, making the strong normalisation proof equivalent to a weak one. Our proof approach is mostly standard, namely induction over types. Although the use of function big-step semantics and environments requires us to define strongly normalising values and environments consisting of strongly normalising values, which are not typically addressed in a standard proof outlined by Abel et al. (2019).

We demonstrate that Sprinkles can be integrated into Kalas, enabling more streamlined choreographic programming with ability to write concrete local computation codes. We also provide typing rules to Kalas enriched with Sprinkles. Our proof results indicate that the strong normalisation property of Sprinkles implies progress for the enriched Kalas. Additionally, the combination of strong normalisation and type soundness of Sprinkles implies type preservation for non-recursive, synchronous transitions in the enriched Kalas. This type preservation result differs from the one by Hirsch and Garg (2022), where the type preservation property of their assumed local language implies type preservation for the choreography.

7.2 Future Work

Immediate future work involves proving type preservation for recursive and asynchronous transitions in the enriched Kalas. Once type preservation is established for all transitions, we can assert

the termination property for a typed choreography: if we have Γ , $\Theta \vdash c \checkmark$, $\mathsf{chorEnvtype}\ \Gamma\ s$, and $\mathsf{chorEnvsn}\ \Gamma\ s$, then there exists a state-choreography pair (s', c') that is related to (s, c) through the reflexive transitive closure of the transition relation defined in Definition 20.

Another immediate task involves updating the scripts for later stages of Kalas's compilation, such as endpoint projection. These scripts may still rely on the original Kalas syntax, so modifications are needed to align them with the changes in syntax and transitions introduced by the enriched Kalas.

In the long term, future work could focus on introducing more message types in the enriched Kalas. Currently, messages are limited to strings, but we can envision processes communicating using richer data types, such as integers, booleans, or even function closures.

Another important area of future work is implementing a more robust exception-handling mechanism for the enriched Kalas. Currently, the choreography transitions directly to $\mathbf{0}$ in response to communication or local computation exceptions. Drawing inspiration from the work of Giallorenzo et al. (2024); Montesi and Peressotti (2017), one could introduce try-catch blocks to manage exceptions and implement recovery strategies. This approach would allow the choreography to handle errors more gracefully, avoiding an immediate transition to $\mathbf{0}$.

Sprinkles uses a functional big-step semantics and looks up values for variables in an environment. It would be interesting as well to implement a small-step semantics with a substitution model for Sprinkles to explore how this change might affect the progress and type preservation proofs for the enriched choreography.

Chapter 8

Test

We define what it is for a choreograph to be well-formed with the $G\ ,\ Th\ \vdash\ c\ \checkmark$ relation.

This is a theorem:

$$\Gamma\ ,\ \Theta\ \vdash\ c\ \checkmark\ \wedge\ \mathsf{chorEnvtype}\ \Gamma\ s\ \wedge\ \mathsf{chorEnvsn}\ \Gamma\ s\ \Rightarrow\ \exists\,\tau\ l\ s'\ c'.\ s \rhd c\ \xrightarrow[l]{\tau}\ s' \rhd c'\ \vee\ \neg\mathsf{not\_finish}\ c$$

$$\mathsf{not\_finish}\ c\ \wedge\ \mathsf{chorEnvsn}\ \Gamma\ s\ \wedge\ \mathsf{chorEnvtype}\ \Gamma\ s\ \wedge\ \Gamma\ ,\ \Theta\ \vdash\ c\ \checkmark\ \wedge\ s \rhd c\ \xrightarrow[l]{\tau}\ s' \rhd c'\ \Rightarrow\ \exists\,\Gamma'.\ \mathsf{chorEnvsn}\ \Gamma'\ s'\ \wedge\ \mathsf{chorEnvtype}\ \Gamma'\ s'\ \wedge\ \Gamma'\ ,\ \Theta\ \vdash\ c'\ \checkmark$$

$$\mathsf{envtype}\ G\ E\ \wedge\ G\ \vdash_s\ e\ :\ ty\ \Rightarrow\ (\exists\,v.\ \mathsf{eval\_exp}\ c\ E\ e\ =\ \mathsf{Value}\ v\ \wedge\ \vdash_s v\ :\ ty)\ \vee\ (\exists\,exn.\ \mathsf{eval\_exp}\ c\ E\ e\ =\ \mathsf{Exn}\ exn)\ \vee\ \mathsf{eval\_exp}\ c\ E\ e\ =\ \mathsf{Timeout}$$

$$\emptyset\ \vdash_s\ e\ :\ t\ \Rightarrow\ (\exists\,cl\ v\ E.\ \mathsf{eval\_exp}\ cl\ E\ e\ =\ \mathsf{Value}\ v\ \wedge\ \mathsf{sn\_v}\ t\ v)\ \vee\ \exists\,cl\ exn\ E.\ \mathsf{eval\_exp}\ cl\ E\ e\ =\ \mathsf{Exn}\ exn$$

$\mathsf{BinOp\ Add}\ (\mathsf{Var}\ x)\ (\mathsf{IntLit}\ 1)$

The transition relation looks like $\mathsf{eval\_exp}\ clk\ E\ exp$

Theorem 5. *some text here*

1. *(Operational completeness) If* $G\ ,\ Th\ \vdash\ c\ \checkmark$ *then there exist ...*
2. *(Operational soundness) If* $\mathsf{eval\_exp}\ clk\ E\ exp$ *then there exist ...*

$$T ::= \mathsf{intT}\ |\ \mathsf{strT}\ |\ \mathsf{boolT}\ |\ \mathsf{fnT}\ t_1\ t_2\ |\ \mathsf{pairT}\ t_1\ t_2\ |\ \mathsf{sumT}\ t_1\ t_2$$

Table 8.1: semantics: communication rules. The function $\mathrm{wv}(\alpha)$ returns the variable (if any) that is modified by α.

$$\textsc{Com}\ \frac{s\ (v_1, p_1)\ =\ \mathsf{StrV}\ d \quad p_1\ \neq\ p_2}{s \rhd p_1.v_1 \mathrel{->} p_2.v_2; C\ \xrightarrow[\epsilon]{p_1.v_1 \gg p_2.v_2}\ s[(v_2, p_2)\ :=\ \mathsf{StrV}\ d] \rhd C}$$

$$\text{T-}\textsc{Var}\ \frac{G\ x\ =\ ty}{G\ \vdash_s\ \mathsf{Var}\ x\ :\ ty}$$

$$\text{T-}\textsc{Fn}\ \frac{G[s\ :=\ t] \vdash_s\ e\ :\ ty}{G\ \vdash_s\ \mathsf{Fn}\ s\ e\ :\ (\mathsf{fnT}\ t\ ty)}$$

$$\text{T-}\textsc{App}\ \frac{G\ \vdash_s\ e_1\ :\ (\mathsf{fnT}\ t\ ty) \quad G\ \vdash_s\ e_2\ :\ t}{G\ \vdash_s\ \mathsf{App}\ e_1\ e_2\ :\ ty}$$

$$\text{CT-}\textsc{Com}\ \frac{\Gamma\ (v_1, p_1)\ =\ \mathsf{strT} \quad \{\ p_1;\ p_2\ \}\ \subseteq\ \Theta \quad p_1\ \neq\ p_2 \quad \Gamma[(v_2, p_2)\ :=\ \mathsf{strT}],\ \Theta\ \vdash\ c\ \checkmark}{\Gamma\ ,\ \Theta\ \vdash\ p_1.v_1 \mathrel{->} p_2.v_2; c\ \checkmark}$$

$$\text{CT-}\textsc{Let}\ \frac{\mathsf{localise}\ \Gamma\ p\ \vdash_s\ e\ :\ ety \quad \Gamma[(v, p)\ :=\ ety],\ \Theta\ \vdash\ c\ \checkmark}{\Gamma\ ,\ \Theta\ \vdash\ \mathsf{Let}\ v\ p\ e\ c\ \checkmark}$$

$$\mathsf{sn\_v\ intT}\ (\mathsf{IntV}\ n)\ \stackrel{\mathsf{def}}{=}\ \mathsf{T}$$
$$\mathsf{sn\_v\ strT}\ (\mathsf{StrV}\ s)\ \stackrel{\mathsf{def}}{=}\ \mathsf{T}$$

```
eval_exp c E (Var str) ≝
 case E str of None ⇒ TypeError | Some v ⇒ Value v
eval_exp c E (Fn s e) ≝ Value (Clos s e (E ⌊fv(e) \\ s))
eval_exp c E (App e1 e2) ≝
 if c > 0 then
  do
   v1 ← eval_exp c E e1;
   v2 ← eval_exp c E e2;
   case v1 of
    IntV v11 ⇒ TypeError
  | StrV v12 ⇒ TypeError
  | BoolV v13 ⇒ TypeError
  | PairV v14 v15 ⇒ TypeError
  | SumLV v16 ⇒ TypeError
  | SumRV v17 ⇒ TypeError
  | Clos s e E1 ⇒ eval_exp (c − 1) E1[s := v2] e
  od
 else Timeout
```

Choreography	External Computation
1. server.var -> client.x;	**fun** $mod\ x\ =$
2. **let** v@client = mod(x) **in**	case Option.map (fn s $\Rightarrow$ valOf (Int.fromString s)) (hd x) of
3. client.v -> server.$result$;	None $\Rightarrow$ None
	| Some n $\Rightarrow$ Some [Int.toString (n MOD y)]

```
1.  server.var -> client.x;
2.  let v@client = StrOf ((NumOf (Var x)) Mod (Var y)) in
3.  client.v -> server.result;
```

Appendix A

Appendix: Explanation on Appendices

Appendix B

Appendix: Explanation on Page Borders

What you find here is an explanation of why the border width keeps flipping from left to right – which you might have spotted and wondered why that's the case.

Firstly, that is *intended* and thus correct, so there is no reason to worry about this. The reason is that this document is configured as a two-sided book, which means:

- We assume the document will be printed out,
- that this will be done in a two-sided mode (i.e., the document will be printed on both sides of each page), and
- that the bookbinding will be in the middle, just like in every book.

When you open the book, there are three borders of equal size n. This however requires that even pages have a border of n on their left and $\frac{n}{2}$ on their right, and odd pages have a border of $\frac{n}{2}$ on their left and n on their right. This is illustrated in Figure B.1.

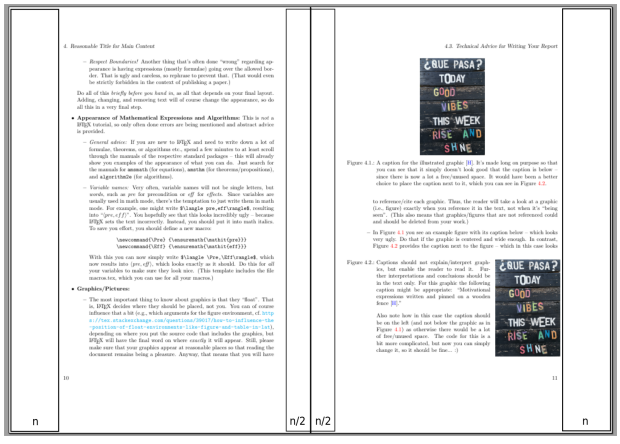

Figure B.1: Illustration showing why page borders flip.

Bibliography

Abel, A.; Allais, G.; Hameer, A.; Pientka, B.; Momigliano, A.; Schäfer, S.; and Stark, K., 2019. Poplmark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming*, 29 (2019), e19. [Cited on pages 11 and 41.]

Carbone, M.; Honda, K.; and Yoshida, N., 2007. Structured communication-centred programming for web services. In *Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007. Proceedings 16*, 2–17. Springer. [Cited on page 5.]

Carbone, M. and Montesi, F., 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. 48, 1 (2013), 263–274. doi:10.1145/2480359.2429101. https://dl.acm.org/doi/10.1145/2480359.2429101. [Cited on pages 1 and 9.]

Cruz-Filipe, L.; Graversen, E.; Lugović, L.; Montesi, F.; and Peressotti, M., 2022. Functional choreographic programming. In *International Colloquium on Theoretical Aspects of Computing*, 212–237. Springer. [Cited on page 10.]

Cruz-Filipe, L. and Montesi, F., 2017. A core model for choreographic programming. In *Formal Aspects of Component Software: 13th International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers 13*, 17–35. Springer. [Cited on pages 1, 2, 6, and 9.]

Giallorenzo, S.; Montesi, F.; and Peressotti, M., 2024. Choral: Object-oriented choreographic programming. *ACM Transactions on Programming Languages and Systems*, 46, 1 (2024), 1–59. [Cited on pages 10 and 42.]

Gordon, M. J., 1988. Hol: A proof generating system for higher-order logic. In *VLSI specification, verification and synthesis*, 73–128. Springer. [Cited on page 6.]

Hallal, R.; Jaber, M.; and Abdallah, R., 2018. From global choreography to efficient distributed implementation. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, 756–763. doi:10.1109/HPCS.2018.00122. https://ieeexplore.ieee.org/abstract/document/8514427?casa_token=B9uMnW0mxFEAAAAA:DMmhwgQZJnHAX6o6p-EHBs4K9rct4pEKen9fdt2CXHC6NOWZxpHT5FSZZFAchGBDjiqmPeviH1U. [Cited on page 1.]

Hardin, T.; Jaume, M.; Pessaux, F.; and Donzeau-Gouge, V. V., 2021. *Concepts and semantics of programming languages 1: a semantical approach with OCaml and Python*. John Wiley & Sons. [Cited on page 17.]

Hirsch, A. K. and Garg, D., 2022. Pirouette: higher-order typed functional choreographies. 6 (2022), 23:1–23:27. doi:10.1145/3498684. https://dl.acm.org/doi/10.1145/3498684. [Cited on pages 1, 9, and 41.]

Kumar, R.; Myreen, M. O.; Norrish, M.; and Owens, S., 2014. Cakeml: a verified implementation of ml. *ACM SIGPLAN Notices*, 49, 1 (2014), 179–191. [Cited on page 6.]

Montesi, F. and Peressotti, M., 2017. Choreographies meet communication failures. *arXiv preprint arXiv:1712.05465*, (2017). [Cited on pages 10 and 42.]

Montesi, F. and Yoshida, N., 2013. Compositional choreographies. In *CONCUR 2013 – Concurrency Theory* (Berlin, Heidelberg, 2013), 425–439. Springer. doi:10.1007/978-3-642-40184-8_30. [Cited on page 1.]

Mullender, S., 1990. *Distributed systems*. ACM. [Cited on page 5.]

Needham, R. M. and Schroeder, M. D., 1978. Using encryption for authentication in large networks of computers. 21, 12 (1978), 993–999. doi:10.1145/359657.359659. https://dl.acm.org/doi/10.1145/359657.359659. [Cited on page 1.]

Owens, S. and Laurent, T., 2016. Prove that simply typed call-by-value lambda calculus programs always terminate. https://github.com/HOL-Theorem-Prover/HOL/tree/develop/examples/fun-op-sem/cbv-lc. Accessed: 2024-10-19. [Cited on page 11.]

Owens, S.; Myreen, M. O.; Kumar, R.; and Tan, Y. K., 2016. Functional big-step semantics. In *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings 25*, 589–615. Springer. [Cited on pages 2, 11, and 13.]

Pierce, B. C., 2002. *Types and programming languages*. MIT press. [Cited on pages 18 and 19.]

Pohjola, J. Å.; Gómez-Londoño, A.; Shaker, J.; and Norrish, M., 2022. Kalas: A verified, end-to-end compiler for a choreographic language. In *13th International Conference on Interactive Theorem Proving (ITP 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. [Cited on pages 1 and 9.]

Slind, K. and Norrish, M., 2008. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, 28–32. Springer. [Cited on pages 2 and 6.]

W3C WS-CDL Working Group, 2005. Web Services Choreography Description Language Version 1.0. Technical report, World Wide Web Consortium (W3C). http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/. Accessed: October 14, 2024. [Cited on page 5.]