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-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 choregraphy 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 choregraphy. 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, synchronised transitions.

Table of Contents

Chapter 1

Introduction

Distributed systems consist of multiple endpoints that communicate by exchanging messages, operating with asynchrony and parallelism as these messages are sent and received between the various endpoints. But programming distributed system is notoriously error-prone as programmer has to implement the communication protocol by developing individual endpoint programs. Mismatched message sending and receiving can lead to errors such as *deadlock*, where the system is waiting forever for a message.

Choreography arises as a programming diagram 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. $\texttt{Alice} \to \texttt{Bob} : \mathit{key}$
2. $\texttt{Bob} \to \texttt{Alice} : \mathit{message}$

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

While most choreography languages focus on the message exchange behaviours, few pay attention to the local computation happening in the individual endpoint. It is shown by Hirsch and Garg (2022) that as long as the local language exhibits type preservation and progress, the choreography inherits these properties as well. Thus, when analyzing message exchange behaviors in choreography—such as multiparty sessions, asynchrony, and parallelism—one can safely assume good behaviours of local computation (Montesi and Yoshida, 2013; Cruz-Filipe and Montesi, 2017; Carbone and Montesi, 2013).

But 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 choregraphy 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 choregraphy and external language lead to cumbersome code.

Thus this thesis extends Kalas, a verified choregraphy 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 codes. The previous choregraphy 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 choregraphy 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 synchronised 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 function closure. According to the approach taken by Owens et al. (2016), we use a functional big-step semantics with clocks 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 convertor is implemented as well for the integration with Kalas. Last but not least, we prove the enriched Kalas enjoys progress. We also show type preservation holds for non-recursive, synchronised transitions in the enriched Kalas.

Chapter 2

Background

2.1 Choreography as a Programming Diagram

A distributed system can be defined as a collection of autonomous computing elements that behaves to its users' expectations (Mullender, 1990). Message exchanges are heart of the distributed system designs cause it allows nodes to collaborate and share resources with each other. Otherwise there is no need to put different nodes inside one connected network. Using a traditional way to describe a distributed system, one will have to give a detailed description of operations at each node in the system. For example, a communication between node A and node B is achieved by A sending the message and B receiving the message. But mismatched message sending and receiving can happen and these endpoint programs may fail to prevent the system from deadlocks or race among messages.

Choreography raises as an effort to eliminate incorrect system implementation by only providing a global description on how messages should be exchanged within the system. This approach is analogous to dance choreography, which outlines steps and movements for an entire performance without focusing on individual dancers' control points.

While it is fascinating to have no mismatched messages in choregraphy perspective, choregraphy cannot be run directly on individual nodes in a system and thus endpoint programs are still required by implementation. Thus the idea of EndPoint Projection (EPP) is proposed in which a choreography is projected into endpoint programs such that each endpoint correctly implements the behaviours described by its role in the choreography. This idea is first described by the design document of Web Services Choreography Description Language (WS-CDL) (W3C WS-CDL Working Group, 2005), and Carbone et al. (2007) further formalise it into the theory of EPP, namely the *soundness* and *completeness* properties for a given EPP. Soundness means that all projected endpoint communications adhere to the choregraphy description and completeness means that all communications described by the choregraphy are projected into endpoint codes.
- maybe two important results but not namely

- idea of CC

- common aspected in choreography: asynchrony ...

2.2 Interactive Theorem Proving

- what is higher-order logic - how proofs are done using HOL - recInduct in HOL - tactices - finite maps?

2.3 Kalas

- overview including intro to its compiler - semantics - implementation in HOL

Chapter 3

Related Work

some intro ..

3.1 Choreography Models

some intro ..

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 choregraphy 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 choregraphy 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 choregraphy implemented as a java library (Giallorenzo et al., 2024). It treats choregraphy as class and EPP as generating role classes from the choregraphy 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 choregraphy into a termination state (nil)

3.2 Some Title

- functional big-step semantics Owens et al. (2016)

- STLC; language with environment semantics - our typing is not unique

- SN for STLC; Weak and strong SN in terms of non-deterministic; my language is deterministic

Chapter 4

Sprinkles: A Simply Typed Lambda Calculus

- we introduce Sprinkles, a language over expressions; Adopting a functional big-step semantics (owens), we define an interpreter that evaluate expressions to values; we evaluate expressions with clocks to ensure termination, thus our evaluation function is total. Expressions in Sprinkles can also be evaluated into exceptions. We include common arithmetic errors such as division by zero, and integer-string conversion errors such as bad formatted strings. The latter is included for the integration with Kalas, since a process may want to perform integer arithmetic based on the received string which does not encode a valid integer.

Sprinkles is deterministic in a sense that if we increase the clocks an expression will be evaluated to the same values or exceptions.

- Sprinkles closely resembles a STLC, but uses a dynamic environment to store values of free variables, so we always evaluate the given expression inside an environment E. This leads to typing expressions in a typing environment G that stores types for free variables. This setup, further requires the typed expression being evaluated in a *correct* environment E in the type soundness proof, given the typing environment G where the expression was typed.

4.1 Syntax

- syntax definition (+ exceptions) with brief explanation
Definition 1 (Sprinkles syntax)**.** Values *and* expressions *in Sprinkles, ranged over by* v, e*, are inductively defined by the grammar*

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

Syntax for function and application syntax are standard. Same as if and let. Case here is for sum types, where *s1*, *e1* represent the left branch and *s2*, *e2* represent the right branch. Var x represents a variable where x is the string that represents variable's name. We include literals for string, integer and boolean in our syntax to allow expressions such as BinOp Add (Var x) (IntLit 1). For the sake of readability, we will write the addition example as Add (Var x) (IntLit 1), and same for any binary or unary operators examples.

We abstract binary and unary operators to allow shorter syntax definitions. This makes extending binary and unary operators easy since we only need to register the new operator in *bop* or *uop* definition and provide the corresponding evaluation rule. It also greatly reduces the number of cases generated in proofs that rely on syntax forms of an expression since we only have one case for all binary operators and one case for all unary operators. Though those two cases typically rely on lemmas that establish the desired properties for all operators, as discussed in Section ?, most operators cases can be proven using the same tactics and where some operator case in the lemma proof differs from the others is usually the essential subproof that is required to re-establish the desired property after introducing this operator to our language. Thus separating concrete operator syntax definitions from language syntax definitions also enabling a neat proof maintenance process when new operators are introduced.

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, Add (IntLit 1) (BoolLit T)), or we reach an exception, or the evaluation times out for the given clock.

4.2 Statics

Our typing rules for expressions are similar to those for a simply typed lambda calculus. We define a typing relation for values as we want the evaluated values in the type soundness statement to be typed as well.

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.

Similar to how we provide only one general case for binary/unary operation evaluation and delegate the evaluation of concrete operators to another function, we provide only one general

rule for typing binary/unary operations and delegate type checking concrete input/output types against a concrete operator to external relations, boptype and uoptype. As we will see in Section ?, this leads to only two cases in the type soundness proof for operators and a separate soundness lemma regarding the operator typing relation is required by the general case.

4.2.1 Syntax and Typing Rules

Definition 2. - *typing relations: typecheck*
Definition 3. - *typing relations: value_type*

- typing relations: typecheck, value_type, uop_type, bop_type

A typing relation for expression is written as (G entails e: ty) where G is the typing environment. For typing a binary operation expression, we require the expressions as arguments to the operator to have some type *t1*, *t2*, and those types and the output type satisfy the typing relation specified in boptype for the concrete binary operator involved. Typing for a unary operation is the same. Other rules are standard.

Before we explain value_type, we define a typing constrain between a typing environment G and a dynamic environment E.
Definition 4. - *envtype definition.*

Intuitively, envtype G E means that values of variables stored in E satisfy the typing constrains specified in G.

value_type relates a value v to a type t. Integer, string and boolean literal cases are obvious. For typing a closure value, we require the function body e to have the output type *t2* in a typing environment G that extends with the type *t1* for function argument s, where such G is a correct descriptions of types of variable values stored in E. This constrain is equivalent as requiring envtype G E. [1]

We require envtype G E is because we will evaluate the function body e in an extended environment E(+s,v) and we want the extended typing environment G(+s,v) in which e is typed to correctly describes the types of values stored in E(+s,v), i.e. envtype $E(+s, v)$ $G(+s, v)$. We will see in Section ? that envtype $E(+s, v)$ $G(+s, v)$ is necessary to be able to use the inductive hypothesis in the application case of type soundness proof as our type soundness statement requires a typed expression to be evaluated in an environment that aligns with the typing environment that types the expression.

value_type for pair and sum cases are also standard. A SumLV v value does not have a unique type since *t2* is arbitrary and so as a SumRV v value.

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

[1] We do not use envtype G E directly in the closure case of value_type because definition of envtype depends on value_type and we want to avoid mutually recursive definition between a definition and a relation in HOL4.

the empty typing environment since f is free in the definition expression. A stuck derivation tree is provided in Appendix ?.

4.3 Dynamics

As discussed above, we defined our semantics using evaluation function rather than relation. The size of expressions are decreasing except for the application case, so we decrease the clock to ensure termination. Semantics for if, let, case are standard. When evaluating a variable, we try to look up for the value corresponds to the variable name in the given environment. If we try to look up an unknown variable name, the evaluation function will return with type error. Environments are implemented as finite maps in HOL4, as discussed in Section ?.
Definition 5. - *semantics rules*

Our evaluation strategy is call-by-value. For evaluating binary or unary operations, we first evaluate the argument expressions into values, and then use another function eval_bop or eval_uop to apply the operators by the specified rules on the evaluated values and return the result. eval_bop and eval_uop are where concrete evaluation rules for each operator is defined.

For example, eval_exp c E (NumOf (Var x)) will first compute eval_exp c E (Var x) by looking up name x in E. If x is unknown to E, the function will return TypeError and propagate it to the top. Otherwise it will return the value v stored under the name x in E and we move to compute eval_uop NumOf v. If v is not a correctly formatted integer string (for example, if it contains a non-numeric character), eval_uop NumOf v will return $ExnBadStr$. Otherwise, it will return the converted integer value IntV n. Again, the format checking and conversion algorithm are implemented in eval_uop's definition. The same applies to evaluating any binary operations. Definitions of eval_uop and eval_bop are included in Appendix ?.

- typed example: eval_exp c $\emptyset$

4.3.1 Function Closure

Definition 6. - *function semantics rule (unrestricted version)*

For a function Fn x e, closure is a triple (x, e, E). x is the function parameter. e is the function body. E is the environment that stores bindings of free variables when the function definition is evaluated.

We stores the bindings of free variables (i.e. the environment) into the closure value when evaluate a function definition so we never lose track of values for bound variables when evaluating a function definition that appears nested in another function definition. If we don't keep track of values for bound variables, when we finishing evaluating a function definition that contains nested function definitions, only the inner most bound variable will appear bound and the inner most function body will be the expression to be evaluated when we apply this nested function definition to a value. Since we now lose bindings for all the other bound variables from outer scopes, we are unable to finish the evaluation of application.

Example 4.3.1. - example: failure without closure

This is illustrated by Example 4.3.1. 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. - another example

Restricted Function Closure

The Kalas repository states an interesting property: Given a bigger state Γ_1 that contains Γ, the choregraphy c still transits into c' and Γ'_1 contains Γ'.
Theorem 1. - *Property theorem.*

However, our S-Fn invalidates this property since a bigger state may result in a closure value that contains a bigger environment, so we will not be able to transit into the same c'. In order to re-establish this property, we modify S-Fn to store a restricted environment instead: (DRESTRICT E (free$_{\mathsf{v}}$ars e) \\ s), where DRESTRICT is defined as:
Definition 7. - *definition of* DRESTRICT

And the modified S-Fn is defined as:
Definition 8. - *restricted eval_exp fn*

The restricted closure should at least contain bindings of variables that appear free from the perspective of the current function definition scope, as much as provided by the environment in which the current function definition is evaluated. Such that if we apply the evaluated restricted function closure to a value, if we were able to evaluate the application without restriction, we will be able to evaluate to the same result with restriction.

But we further require the restricted environment to exclude storing the value of the variable bound by current function definition in the current environment if any. This still allows us to evaluate an application to the same result because if we ever apply this function to a value we will always bind s with the provided value, leaving the previous value of s stored in the environment irrelevant.

Excluding storing s is also necessary. If we only restrict E with free$_{\mathsf{v}}$ars e, since e can have s as free variables and Γ does not need to contain binding for s for evaluating Fn s e, (DRESTRICT (localise Γ p) (free$_{\mathsf{v}}$ars e)) may not contain binding for s. But since Γ' contains Γ, Γ' may contain binding for s and thus (DRESTRICT (localise Γ' p) (free$_{\mathsf{v}}$ars e)) contains binding for s, making the closure evaluated in a bigger state larger than the closure evaluated in the smaller state.

The restricted S-Fn still evaluates Example 4.3.2 to the same result, as illustrated in Example 4.3.3. When the inner application is evaluated, the parameter x of the current function is assigned with the evaluated argument value 1 in the environment. Since x, which is bound by the outer function definition, is a free variable from the perspective of the inner function definition, the binding of x is stored into closure evaluated in line 5. So we are able to evaluate $x + y$ to 3 in line 8.

Example 4.3.3. - restricted currying evaluation example

Excluding the parameter and storing only bindings of free variables into the closure environment also makes sense in a way that bindings of free variables are necessary to evaluate the function body, and we will always have an evaluated value for the function parameter when the application is evaluated regardless of whether the closure environment contains a previous binding of the function parameter or not.

Another possible approach to re-establish Theorem 1 is to define an equivalence relation on function closures where closures with the same parameter, function body are equivalent if their environments all contain the same bindings of free variables in the function body, excluding by not considering the binding of function parameter if any. This leads to a more complex approach since one has to deal with choreographies that contains different but equivalent function closures. Though our approach modifies the S-Fn rule and requires extra lemmas to prove the function case in the type soundness proof (discussed in Section 4.4.1), it is simpler since given a bigger state the function definition will still be evaluated to the same closure and no discussion on equivalence relation is required. If not specified, the restricted S-Fn will be used for the rest of the thesis.

Recursion

Sprinkles does not support recursion, as we see in the Example ?, the name f is unbound in the definition thus we don't have a value for the recursively invoked name f in line 6.

- example

We argue that recursion can be added to Sprinkles by introducing Letrec syntax to our expressions, and Letrec f s $e1$; $e2$ will be evaluated into a quadruple (f, x, e, E). The quadruple is similar to the one described by Hardin et al. (2021).

4.3.2 Main Properties

The total evaluation function obtained by using a functional big-step semantics makes Sprinkles's evaluation deterministic. We say that the evaluation for an expression *terminates* if there exists a clock with which the evaluation function return a result that is not Timeout. Otherwise if for all clocks the evaluation for an expression returns Timeout, we say the evaluation for this expression *diverges*. Then the evaluation of Sprinklesis deterministic in a sense that if the evaluation for an expression terminates for an initial clock, then if we increase the clock we will always evaluate to the same result.

Theorem 2. - *theorem*

Theorem 2 is important in the strong normalisation proof. For example, in the if case of the proof, we know evaluation for boolean guard and two branches terminates by the inductive hypothesis on their separate clock, we want to ensure that there exists a common clock on which the evaluation of all three expressions also terminate and return the same results. More discussion on the use of Theorem ? in strong normalisation proof can be found in Section ?.

4.4 Type Soundness

some intro ...

4.4.1 Lemmas

We discuss several lemmas and propositions for the proposed semantics and typing rules before attempting 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.
Lemma 1. - *typecheck_*_thm: the generation lemma*

Proof: Immediately follows from the typecheck definition. Box

The generation lemma allows us to move one step forward in the type derivation tree in the bottom to top direction because it tells us how the current type was derived. We can obtain types or typing information of subterms in a well-typed term.
Lemma 2. - *valuetype_EQ_*: the lemma of canonical forms of types (only useful cases)*

Proof: Immediately follows from the value_type definition. Box.

This gives us the concrete syntax of a value v and typing information for terms involved in the concrete syntax if we have value_type v t. This is useful for inductive cases in the type soundness proof, since we will have expressions evaluated to a value v and value_type v t by inductive hypothesis. And the concrete syntax of v matches terms in our goal.

The type preservation under substitution lemma used by Pierce (2002) says substituting variables with term of appropriate types preserves the original type. Since we evaluate variables into values by looking up in an environment E, envtype G E (Definition ?) is analogous to the type preservation under substitution lemma.

As discussed in Section ? and Section ?, we evaluate the argument expressions into values first and delegate the evaluation of binary/unary operation to eval_bop and eval_uop. We also type the argument expressions first and then delegate checking input/output types against the concrete operator to a separate typing relation boptype and uoptype. This leads to two important properties: if argument values are typed and the operator types with the argument types, eval_bop and eval_uop will always output a value of correctly type or we reach an exception. We refer those 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 2. Box
Proposition 2. - *uop soundness*

Proof: Same as the above. Box

Proposition 1 and Proposition 2 are needed for the binary and unary operator cases in the type soundness proof. As we will have a well-typed evaluated argument values by the inductive hypothesis and we need to show eval_bop and eval_uop output a well-typed value or 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 3. - *envtype_lemma*

Proof: Since value_type (Definition ?) is inductively defined, we prove by rule induction over value_type. Integer, string, boolean literals are base cases and directly follows the definition of

envtype (Definition ?). Pair and sum values cases follow directly from the inductive hypothesis. The closure value case does not require using inductive hypothesis, but rather follows the closure case of the lemma of canonical forms of types (Lemma 2). Box

This lemma is useful in the function, application and sum case cases of the type soundness proof, where we have subterms inside the goal expression typed in an extended typing environment and we have to re-establish the envtype property for the extended typing environment and dynamic environment so we can use the results from corresponding inductive hypothesis.

Restricted Function Closure

If we adopt the restricted S-Fn semantics (discussed in Section 4.3.1), we will need the following lemmas so we can type the closure value that contains a more restricted dynamic environment Clos x e (DRESTRICT$Efv(e)deletes$) given that the expression of function body e is typed in an un-restricted extended typing environment $G(+x,t)$. However, the following lemmas hold regardless of whether we adopt a restricted function closure. They are simply unnecessary for the type soundness proof when using an unrestricted function closure.

The first lemma says that if en expression e is typed in an environment G then G contains type information for all free variables in e.
Lemma 4. - *typecheck_env_fv*

Proof: By rule induction on typecheck and rewriting the definition of subset relation and the commutativity property of subset relation. Box.

The second lemma says that if en expression e is typed in an environment $G1$, then e has the same type in a smaller typing environment $G2$, given $G2$ contains type information for all free variables in e.
Lemma 5. - *typecheck_env_submap*

Proof: By rule induction on typecheck and results in the inductive hypothesis match the goal. We define a typing environment G in HOL4 as a finite map that maps a finite set of name strings to a finite set of types. Thus proof for most cases is re-establishing the subset relation between free variables of an expression and the domain of a finite map, and the submap relation between finite maps, in order to use results in the inductive hypothesis. Box.

The third lemma says that we can always type the expression e in a smaller typing environment as long as the current typing environment still contains the bindings for free variables of e.
Lemma 6. - *typecheck_drestrict*

Proof: Follows directly from Lemma 4 and Lemma 5. Box

Lemma 6 and Lemma 4 gives us the existence of a minimal typing environment for an expression e, which contains exactly bindings for all free variables of e.
Corollary 1. - *minimal typing env*

Proof: By Lemma 6, we know that if e is typable, i.e. there exists some G such that $G \vdash_s e : ty$, then we have DRESTRICT G (free$_\mathsf{v}$ars e) $\vdash_s e : ty$. We know that G contains bindings for all free variables in e by Lemma 4, thus DRESTRICT G (free$_\mathsf{v}$ars e) contains exactly bindings for all free variables in e, by Definition 7. For the other direction, if there exists a typing environment G' that contains less bindings than DRESTRICT G (free$_\mathsf{v}$ars e) and we have $G' \vdash_s e : ty$, by Lemma 4 G' contains bindings for all free variables in e. Contradiction. Box

4.4.2 Type Soundness Proof

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

Theorem 3.

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

Proof: We prove via rule induction on the definition of eval_exp. Var case follows directly from Definition 4. For Let $s\ e_1\ e_2$ case, by Lemma 1, we have $G\ \vdash_s\ e_1\ :\ t_0$ and $G[s\ :=\ t_0]\vdash_s\ e_2\ :\ t$ given $G\ \vdash_s\ \mathsf{Let}\ s\ e_1\ e_2\ :\ t$. For all the following cases, we assume Lemma 1 is applied first. Then by the inductive hypothesis for *e1* we know *e1* is evaluated to some value v and value_type $v\ t_0$. Lemma 3 then gives us envtype $G[s\ :=\ t_0]\ E[s\ :=\ v]$. This allows us to use the results in the induction hypothesis for *e2* which prove the goal.

For Fn $s\ e$ case, given envtype $G\ E$ and $G[s\ :=\ t_1]\vdash_s\ e\ :\ t_2$, we need to show value_type (Clos $s\ e$ (DRESTRICT E (free$_v$ar
By the definition of value_type (Definition 3) and envtype (Definition 4), we need to show there exists some typing environment *G0* such that $G_0[s\ :=\ t_1]\vdash_s\ e\ :\ t_2$ and envtype G_0 (DRESTRICT E (free$_v$ars e) \\ s). By Corollary 1 we know there exists a minimal G' such that $G'[s\ :=\ t_1]\vdash_s\ e\ :\ t_2$ and it contains bindings for all free variables of e, excluding binding for s if $s\ \in$ free$_v$ars e. Then by the definition of DRESTRICT (Definition 7) and Definition 4 we have found such *G0* which is G'.

if we use the unrestricted S-Fn, we need to show value_type (Clos $s\ e\ E$) (fnT $t_1\ t_2$) instead, which is straight forward given Definition 3.

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, from the inductive hypothesis for the guard expression e we know e is evaluated to value v and value_type v boolT. By Lemma 2 we know v is either true or false. And we have true and false branches all covered by the inductive hypotheses.

For App $e_1\ e_2$ case, by the inductive hypothesis for *e1* and *e2* we know they are evaluated to *v1* and *v2* separately. We also have value_type v_1 (fnT $t_1\ t_2$) and value_type $v_2\ t_1$. By Lemma 2 we know *v1* is a function closure Clos $s\ e\ E$ which matches S-App in the goal. Lemma 2 also gives us $G[s\ :=\ t_1]\vdash_s\ e\ :\ t_2$ in which envtype $G\ E$. By Lemma 3, we have envtype $G[s\ :=\ t_1]\ E[s\ :=\ v_2]$. This allows us to apply the inductive hypothesis for e which solves the goal.

For Case $e\ s_1\ e_1\ s_2\ e_2$ case, we have e evaluated to value v and value_type v (sumT $t_1\ t_2$) by the inductive hypothesis. By Lemma 2 we know v is either SumLV v' or SumRV v' for some value v'. In both cases we either have value_type $v'\ t_1$ or value_type $v'\ t_2$. Then by Lemma 3 we have either envtype $G[s_1\ :=\ t_1]\ E[s_1\ :=\ v']$ or envtype $G[s_2\ :=\ t_2]\ E[s_2\ :=\ v']$. This allows us to apply the inductive hypothesis for *e1* or *e2* which solves the goal. Box

Chapter 5

Strong Normalisation

- why it is necessary - the halting problem discussion.

- our language is deterministic with the clock lemma, so strong normalisation is the same as weak normalisation

- issue with directly proving: not strong enough IH for the applied term for app case, the applied term may get bigger, cause the e from closure value may be bigger; same issue for a substitution model in (Pierce, 2002) - so sn_v says for a closure to be in sn_v, for any v (argument to function) that in sn_v ("halts"), the evaluation of function body applying v halts

- induction on typing rules/types (e.g. from t',t to t-¿t')

- fn case: sn_e for a function expression of function types, transforms into sn_v funcion types function value (closure) by rewriting the evaluation; which is satisfied by IH — 1. IH says for every dynamic environment that nicely matches the updated typing environment, sn_e t' e

Chapter 6

The Enriched Choreography

6.1 Syntax

6.2 Semantics

- exceptions

6.3 Typing

6.3.1 Typing Rules

6.3.2 Main Properties

6.3.3 Type Soundness

Chapter 7

Concluding Remarks

If you wish, you may also name that section *"Conclusion and Future Work"*, though it might not be a perfect choice to have a section named "A & B" if it has subsections "A" and "B". Also note that you don't necessarily have to use these subsections; that also depends on how much content you have in each. (E.g., having a section header might be odd if it contains just three lines.)

7.1 Conclusion

This section usually summarizes the entire paper including the conclusions drawn, e.g., did the developed techniques work? Maybe add why or why not. Also don't hold back on limitations of your work; it shows that you understood what you have done. And science isn't about claiming how great something is, but about objectively testing hypotheses. Also note that every single scientific paper has such a section, so you can check out many examples, preferably at top-tier venues, e.g., by your supervisor(s).

7.2 Future Work

- asynchronus messages; confluence property - Progress for EPP

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:

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

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

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

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

BinOp Add (Var x) (IntLit 1)

The transition relation looks like eval_exp *clk E exp*

Theorem 4. *some text here*

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

Definition 9 (Kalas syntax)**.** Choreographies *in Kalas, ranged over by* C*, are inductively defined by the grammar*

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

Definition 10 (Sprinkles syntax)**.** Values *and* expressions *in Sprinkles, ranged over by* v, e*, are inductively defined by the grammar*

$$
\begin{array}{rcl}
v & ::= & \mathsf{IntV}\ n \mid \mathsf{StrV}\ s \mid \mathsf{BoolV}\ b \mid \mathsf{Clos}\ s\ e\ E \mid \mathsf{PairV}\ v_1\ v_2 \mid \mathsf{SumLV}\ v \mid \mathsf{SumRV}\ v \\
bop & ::= & \mathsf{Add} \mid \mathsf{Concat} \mid \mathsf{Mult} \mid \mathsf{Div} \mid \mathsf{Mod} \mid \mathsf{Less} \mid \mathsf{And} \mid \mathsf{Or} \mid \mathsf{Eq} \mid \mathsf{Sub} \mid \mathsf{Pair} \\
uop & ::= & \mathsf{Not} \mid \mathsf{NumOf} \mid \mathsf{StrOf} \mid \mathsf{Fst} \mid \mathsf{Snd} \mid \mathsf{SumL} \mid \mathsf{SumR}
\end{array}
$$

$$
\begin{array}{rcllll}
e & ::= & \mathsf{Var}\ x & \textit{(var)} & \mathsf{StrLit}\ s & \textit{(str)} \\
 & & \mathsf{IntLit}\ n & \textit{(int)} & \mathsf{BoolLit}\ b & \textit{(bool)} \\
 & & \mathsf{BinOp}\ bop\ e_1\ e_2 & \textit{(bop)} & \mathsf{Uop}\ uop\ e & \textit{(uop)} \\
 & & \mathsf{If}\ bg\ e_1\ e_2 & \textit{(if)} & \mathsf{Let}\ x\ e_1\ e_2 & \textit{(let)} \\
 & & \mathsf{Fn}\ x\ e & \textit{(fn)} & \mathsf{App}\ e_1\ e_2 & \textit{(app)} \\
 & & \mathsf{Case}\ e\ x\ e_1\ y\ e_2 & \textit{(case)} & &
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{sn_v}\ \mathsf{intT}\ (\mathsf{IntV}\ n) \stackrel{\mathsf{def}}{=} \mathsf{T} \\
\mathsf{sn_v}\ \mathsf{strT}\ (\mathsf{StrV}\ s) \stackrel{\mathsf{def}}{=} \mathsf{T}
\end{array}
$$

```
eval_exp c E (Var str) =def
 case E str of None ⇒ TypeError | Some v ⇒ Value v
eval_exp c E (Fn s e) =def
 Value (Clos s e (DRESTRICT E (free_vars e) \\ s))
eval_exp c E (App e1 e2) =def
 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 $\twoheadrightarrow$ `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 $\twoheadrightarrow$ `server.`$result$;	None ⇒ None
	| Some n ⇒ Some [Int.toString (n MOD y)]

Table 8.1: semantics: communication rules. The function wv(α) returns the variable (if any) that is modified by α.

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

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

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

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

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

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

```
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.

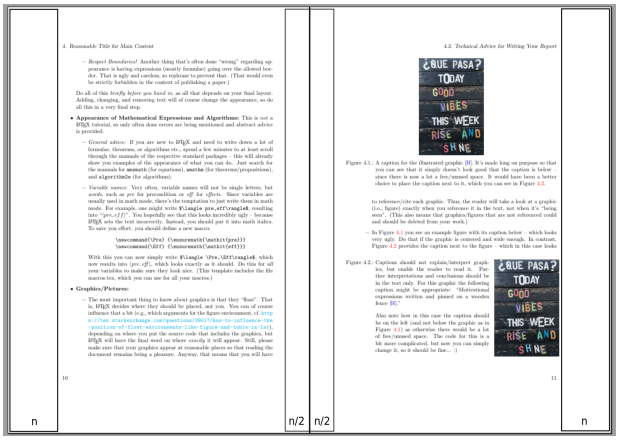

Figure B.1: Illustration showing why page borders flip.

Bibliography

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 7.]

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 8.]

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, and 7.]

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 page 8.]

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 16.]

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 and 8.]

Montesi, F. and Peressotti, M., 2017. Choreographies meet communication failures. *arXiv preprint arXiv:1712.05465*, (2017). [Cited on page 8.]

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.; 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 and 9.]

Pierce, B. C., 2002. *Types and programming languages*. MIT press. [Cited on pages 17 and 21.]

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 7.]

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 page 2.]

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.]