The Australian National University
2600 ACT | Canberra | Australia

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):

$$\begin{array}{ll} \texttt{1.} & \texttt{Alice} \rightarrow \texttt{Bob} : \mathit{key} \\ \texttt{2.} & \texttt{Bob} \rightarrow \texttt{Alice} : \mathit{message} \end{array}$$

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 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 $\twoheadrightarrow$ `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 $\twoheadrightarrow$ `server`.$result$;	None $\Rightarrow$ None
	| Some $n\ \Rightarrow$ 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 $\twoheadrightarrow$ `client`.x;
2. **let** v@`client` = StrOf ((NumOf (Var x)) Mod (Var y)) **in**
3. `client`.v $\twoheadrightarrow$ `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

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

The Environment Model

4.1 Syntax

- binary operators - unary operators (StrOf, NumOf) - value script?

4.2 Semantics

- how the recursion is not supported in our semantics; we don't support fixed point (as in the abstraction book which can't be cited)

- evaluation strategy is defined by the evaluation rules: call by value as in ... by Owens et al. (2016)

- closure modification based on the submap property for choregraphy

- functional big step semantics: intepreter style with clock, total function. for being implemented in HOL4 - properties: clock increment (cases on result) (and clock decrement) - closure: issues with dynamic environment, so we use lexical environment; do we explain restrciting to fv(e) and how? - exceptions

4.3 Typing

4.3.1 Syntax

- rich data types: int, string, boolean, sum type, pair type, closure, ... - typecheck - uoptype, boptype - valuetype - envtype

4.3.2 Typing Rules

- typecheck - uoptype, boptype - valuetype: closure - no type uniqueness

- maybe an example of what program typing rules allow and which disallow and it is actually meaningful

4.3.3 Main Properties

- typecheck_*_thm: the inversion (generation) lemma, (Pierce, 2002); we can calculate the types of subterm of a well-typed term; all automatically handled by HOL4, for matching the subgoals (when no IH), OR for using the IH

- valuetype_EQ_* lemma: canonical forms of types (Pierce, 2002), used for type soundness proof, where we only have v and we don't have the form of v; only needed for boolean value, function value and sum value, where those are intermediate evaluated values in semantics and after the IH has been applied and we want to have a canonical form of v rather than merely the name v itself, because we want the results from IH to match the evaluation semantics in goal, which operates on concrete value forms rather than a name of the value; it's for using the IH results (to match the evaluation process in goal)

- If no intermediate evaluated values are involved in the semantics, we don't have to use IH since no recursive calls to evaluation thus no IH to use; in this case 'irule value_type_*' moves one step higher in the proof tree to the previous sub-proof tree, transforming the goal; we will have the concrete form of values in goal since we rewrite the evaluation definition in our goal

- envtype: analogous to the "Preservation of types under substitution" lemma in a substitution semantics type soundness proof (Pierce, 2002); our equivalent version is by envtype; so substitution gives free variables meaning, we do this by FLOOKUP (var case using envtype_def; other that contains intermediate value evaluation using envtype update lemma)

- bop type soundness and uop type soundness is just the same as operator case in the type soundness proof for a simply typed lambda calculus

- operators: uoptype soundness and boptype soundness (use the invertability) - typecheck: reducing typing environment (for soundness fn case)

4.3.4 Type Soundness

- fn case: - envtype_DRESTRICT - typecheck_update_sub_fv: which needs typecheck_drestrict (which needs typecheck_env_fv), a strengthening on the typing context

- other cases: mostly need envtype_lemma to ensure the updated environments still has envtype

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:

$$\begin{array}{l} \vdash \Gamma\ ,\ \Theta\ \vdash\ c\ \checkmark\ \wedge\ \mathsf{chorEnvtype}\ \Gamma\ s\ \wedge\ \mathsf{chorEnvsn}\ \Gamma\ s\ \Rightarrow \\ \quad \exists\,\tau\ l\ s'\ c'.\ s \rhd c\ \underset{l}{\overset{\tau}{\rightarrow}}\ s' \rhd c'\ \vee\ \neg\mathsf{not\_finish}\ c \end{array}$$

$$\begin{array}{l} \vdash \mathsf{not\_finish}\ c\ \wedge\ \mathsf{chorEnvsn}\ \Gamma\ s\ \wedge\ \mathsf{chorEnvtype}\ \Gamma\ s\ \wedge \\ \quad \Gamma\ ,\ \Theta\ \vdash\ c\ \checkmark\ \wedge\ s \rhd c\ \underset{l}{\overset{\tau}{\rightarrow}}\ s' \rhd c'\ \Rightarrow \\ \quad\ \exists\,\Gamma'. \\ \quad\quad \mathsf{chorEnvsn}\ \Gamma'\ s'\ \wedge\ \mathsf{chorEnvtype}\ \Gamma'\ s'\ \wedge \\ \quad\quad \Gamma'\ ,\ \Theta\ \vdash\ c'\ \checkmark \end{array}$$

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

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

The transition relation looks like $\mathsf{eval\_exp}\ clk\ E\ exp$
Theorem 8.0.1. *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 ...*

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

$$
\begin{array}{lll}
C & ::= & p_1.v_1 \mathrel{->} p_2.v_2; C \quad \textit{(com)} \\
 & & \mathsf{Let}\ v\ p\ e\ C \quad \textit{(let)} \\
 & & \ldots
\end{array}
$$

Definition 2 (Sprinklessyntax)**.** Expressions in Sprinkles, ranged over by e, are inductively defined by the grammar

$$
\begin{array}{llll}
e & ::= & \mathsf{Var}\ x & \textit{(var)} \\
 & & \mathsf{BinOp}\ bop\ e_1\ e_2 & \textit{(bop)} \\
 & & \mathsf{Uop}\ uop\ e & \textit{(uop)} \\
 & & \mathsf{If}\ bg\ e_1\ e_2 & \textit{(if)} \\
 & & \mathsf{Let}\ x\ e_1\ e_2 & \textit{(let)} \\
 & & \mathsf{Fn}\ x\ e & \textit{(fn)} \\
 & & \mathsf{App}\ e_1\ e_2 & \textit{(app)} \\
 & & \mathsf{Case}\ e\ x\ e_1\ y\ e_2 & \textit{(case)}
\end{array}
$$

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

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

$$
\begin{array}{l}
\mathsf{eval\_exp}\ c\ E\ (\mathsf{Var}\ str) \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{case}\ E\ str\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{TypeError} \mid \mathsf{Some}\ v \Rightarrow \mathsf{Value}\ v \\
\mathsf{eval\_exp}\ c\ E\ (\mathsf{Fn}\ s\ e) \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{Value}\ (\mathsf{Clos}\ s\ e\ (\mathsf{DRESTRICT}\ E\ (\mathsf{free_vars}\ e)\ \backslash\backslash\ s)) \\
\mathsf{eval\_exp}\ c\ E\ (\mathsf{App}\ e_1\ e_2) \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{if}\ c > 0\ \mathsf{then} \\
\quad\quad \mathsf{do} \\
\quad\quad\quad v_1 \leftarrow \mathsf{eval\_exp}\ c\ E\ e_1; \\
\quad\quad\quad v_2 \leftarrow \mathsf{eval\_exp}\ c\ E\ e_2; \\
\quad\quad\quad \mathsf{case}\ v_1\ \mathsf{of} \\
\quad\quad\quad\quad \mathsf{IntV}\ v_{11} \Rightarrow \mathsf{TypeError} \\
\quad\quad\quad \mid \mathsf{StrV}\ v_{12} \Rightarrow \mathsf{TypeError} \\
\quad\quad\quad \mid \mathsf{BoolV}\ v_{13} \Rightarrow \mathsf{TypeError} \\
\quad\quad\quad \mid \mathsf{PairV}\ v_{14}\ v_{15} \Rightarrow \mathsf{TypeError} \\
\quad\quad\quad \mid \mathsf{SumLV}\ v_{16} \Rightarrow \mathsf{TypeError} \\
\quad\quad\quad \mid \mathsf{SumRV}\ v_{17} \Rightarrow \mathsf{TypeError} \\
\quad\quad\quad \mid \mathsf{Clos}\ s\ e\ E_1 \Rightarrow \mathsf{eval\_exp}\ (c - 1)\ E_1[s := v_2]\ e \\
\quad\quad \mathsf{od} \\
\quad \mathsf{else}\ \mathsf{Timeout}
\end{array}
$$

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

1. $\texttt{server}.var \mathrel{->} \texttt{client}.x$;
2. **let** v@$\texttt{client}$ = StrOf ((NumOf (Var x)) Mod (Var y)) **in**
3. $\texttt{client}.v \mathrel{->} \texttt{server}.result$;

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

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

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

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

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

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

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

Appendix A

Appendix: Explanation on Appendices

You may use appendices to provide additional information that is in principle relevant to your work, though you don't want *every reader* to look at the entire material, but only those interested.

There are many cases where an appendix may make sense. For example:

- You developed various variants of some algorithm, but you only describe one of them in the main body, since the different variants are not that different.
- You may have conducted an extensive empirical analysis, yet you don't want to provide *all* results. So you focus on the most relevant results in the main body of your work to get the message across. Yet you present the remaining and complete results here for the more interested reader.
- You developed a model of some sort. In your work, you explained an excerpt of the model. You also used mathematical syntax for this. Here, you can (if you wish) provide the actual model as you provided it in probably some textfile. Note that you don't have to do this, as artifacts can be submitted separately. Consult your supervisor in such a case.
- You could also provide a list of figures and/or list of tables in here (via the commands `\listoffigures` and `\listoftables`, respectively). Do this only if you think that this is beneficial for your work. If you want to include it, you can of course also provide it right after the table of contents. You might want to make this dependent on how many people you think are interested in this.

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.

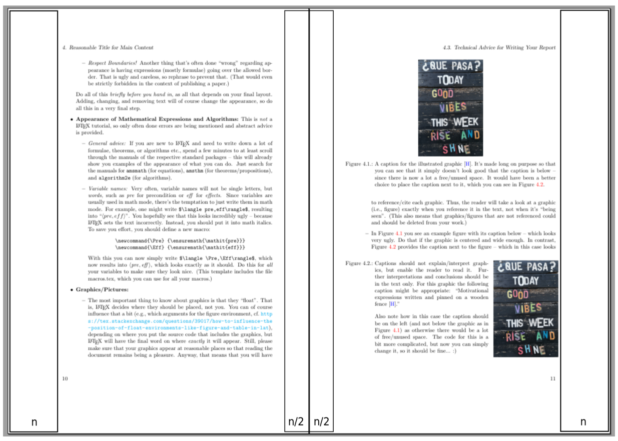

Figure B.1: Illustration showing why page borders flip.

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=B9uMnWOmxFEAAAAA:DMmhwgQZJnHAX6o6p-EHBs4K9rct4pEKen9fdt2CXHC6NOWZxpHT5FSZZFAchGBDjiqmPeviH1U. [Cited on page 1.]

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, 9, and 11.]

Pierce, B. C., 2002. *Types and programming languages*. MIT press. [Cited on pages 12 and 13.]

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.]