Research Internship: Investigating Audited Computation within Dependent Session Types

Author: Christopher Worthington s3715086

Supervisor: J.A. Perez Parra, Prof Dr

Abstract

In this project two areas of research, both rooted in the Curry-Howard correspondence, are investigated in order to potentially combine their different abilities for computational expression. The first area that is explored is dependent session types for concurrent computation where π-calculus processes are strictly typed along with the functional terms that they communicate. The second area uses Justification Logic to type λ-calculus functions then also record and inspect their computational history. An executive summary of both areas is given and then followed up with a speculative summary on how the two can be combined into a system where typed π-calculus processes can communicate audited functional terms.

1 Introduction

Formal systems for modelling of computation can be a convenient tool to check the correctness, complexity, and performance of computer functions and processes [1]. These formal systems each employ a specification of syntax and semantics to represent and reason about real world problems such as logic or software systems. While it is useful to model the programs we know, it is also often the case that computer processes will need to interact with other external processes at run time and it should be possible to check the validity of these sources. There are multiple ways in which these sources can be proven to be trustworthy. Functions and services can be given types that more strictly define their behaviour [2] and these types can be proven and enforced when multiple processes interact with each other. Processes can also communicate separate proofs along with each communication that assert properties of the data being passed [3]. The proofs themselves can also have their own types.

This theory of types for functional and concurrent processes is enabled by the Curry-Howard correspondence [4]. This correspondence is a connection that was discovered between two seemingly disconnected areas of research. It has resulted in a new set of formal representations that can act as both proof systems and modelled programming languages at the same time. The correspondence links programming theory and proof theory in such a a way that logical propositions can be isomorphically equated to program types, proofs to programs, and proof reductions to execution steps of programs [5]. These Curry-Howard isomorphisms are the central tool used by the different languages and systems, that are explored in this paper, when it comes to modelling various programming constructs. They allow for more expression in not only their behaviour, but also the information they use and communicate.

In [6], history based access control is explored as a way to base the current permissions of a process on its previous computations. This is modelled for λ-calculus functions using Justification Logic in [7]. This system also utilises a Curry-Howard isomorphism for functional terms. As stated

previously, communicating processes can enforce some type theory on the information they pass between them. Since these term types could be limited to a specific class of functional terms, it suggests that the audited functional terms could be used for this. This opens the door to processes that can verify the computational history of sequential programs, but in a concurrent setting.

The aim of this project and paper is to investigate how history based access control is modelled in the sequential setting, creating a high level outline in the process. This investigation will then be used to speculate and initially model how history based access control could be utilised for communicating processes, using dependent type theory. In Section 2 an outline of type theory and subsequently its use for π-calculus is given. Section 3 has a more in depth summary of how Justification Logic is used for audited computation leading to the definition of $\lambda^{\mathfrak{h}}$. Following this, Section 4 investigates how these functions paired with their computation history could be used within π-calculus processes.

2 Type Theory for Computation

2.1 Linear Types

The first system to consider comes in [8] which connects linear logic, a logic with controlled use of resources, with λ-calculus, a process calculus for sequential computation. Linear logic adds propositions that are restricted to limited use in reasoning. So if we use a linear proposition A to deduce another proposition B through linear implication, now written as $A \multimap B$, then A is no longer a proposition we can use. This deduction is represented by the implication elimination rule in Figure 1. Linear logic uses a natural deduction proof system. Here the judgement $\Gamma \vdash A \multimap B$ tells us that given the assumptions in Γ, we can consume A and yield B. Then given the other judgement of $\Delta \vdash A$, showing that we have A from assumptions in Δ, the deduction step can be taken to obtain $\Delta \vdash B$. In this deduced judgement we still have the assumptions in Δ but since have B, we no longer have A.

$$\frac{\Gamma \vdash A \multimap B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B} \multimap -E$$

Figure 1: $\multimap$-E Rule for Linear Logic

This logic lends itself perfectly to functional programming as in most situations, when a computation step takes place, the previous state of the program is not retained but replaced by the new state. A Curry-Howard isomorphism is drawn here in order to build a constructive proof system that serves a purpose for both linear logic and for the functional λ-calculus. Functions are defined as terms that uniquely encode proofs for linear propositions, and linear propositions specify the behaviour of λ-calculus functions. These linear typed terms have intuitionistic judgements of the form $\Gamma \vdash s : A$ which reads as "Term s has type A given assumptions Γ" or "Proposition A is true from proof s given assumptions Γ". The term s is a λ-calculus function and the type A is a linear logic proposition. The assumptions in Γ are either the form $\langle s : A \rangle$ or $[s : A]$ depending on if they are linear (usable once) or intuitionistic (usable any number of times) respectively.

$$\frac{\Gamma \vdash s : A \multimap B \quad \Delta \vdash t : A}{\Gamma, \Delta \vdash s \cdot t : B} \multimap -E$$

Figure 2: $\multimap$-E Rule for Linear Types

All valid types can then be derived using construction rules and axioms, an example of one is shown in Figure 2. It is isomorphic to the rule in Figure 1 but is now enhanced with the λ-calculus proof terms. Linear implication $A \multimap B$ now represents a function s that accepts an input of type A and then continues as type B. Given that the function t has type A, s applied to t ($s \cdot t$) then has type B. The two contexts Γ and Δ are also combined in the new context to maintain the typing of s and t. This is equivalent to how previously A was consumed in $A \multimap B$ to obtain B. Now t is consumed by s to obtain $s \cdot t$. The judgement that results in Figure 2 also uniquely encodes the proof as the $\cdot$ operator will only appear in the case of this reduction step. This is the case for all deduction steps and λ-calculus terms. So given a judgement of the form $\Gamma \vdash s : A$, the entire proof tree can be derived.

The last part to a Curry-Howard isomorphism is the link between proof reduction and computational reduction. An example of a proof reduction for linear logic is shown in Figure 3. This normalisation step eliminates the unnecessary application of $\multimap$-E and $\multimap$-I as with $\Delta \vdash A$ and the steps contained in proof u; $\Gamma, \Delta \vdash B$ can be deduced directly. Now if we were to apply Linear Types here, the judgement $\Gamma, \Delta \vdash B$ will be typed differently in these two instances due to the fact that the λ-calculus terms uniquely encode each proof. For the proof on the left we would have a judgement of the form $\Gamma, \Delta \vdash (\lambda x.u) \cdot t : B$. But without the application of $\multimap$-E or $\multimap$-I, we instead have a judgement of the form $\Gamma, \Delta \vdash u : B$ (where occurrences of x in u have been replaced by t). This reduction then directly corresponds to the computational reduction $(\lambda x.u) \cdot t \implies u[t/x]$

$$\dfrac{\dfrac{\begin{array}{c}\langle A \rangle \vdash A \\ \vdots u \\ \Gamma, \langle A \rangle \vdash B\end{array}}{\Gamma \vdash A \multimap B} \multimap\text{-I} \qquad \begin{array}{c}\vdots t \\ \Delta \vdash A\end{array}}{\Gamma, \Delta \vdash B} \multimap\text{-E} \quad \Longrightarrow \quad \begin{array}{c}\vdots t \\ \Delta \vdash A \\ \vdots u \\ \Gamma, \Delta \vdash B\end{array}$$

Figure 3: Linear Logic Reduction Rule

2.2 Dependently-Typed Sessions for π-calculus

π-calculus terms represent processes which can synchronise and communicate via named channels. Like how the Curry-Howard correspondence was used for λ-calculus functions, it is also applied to use linear logic again but instead with a fragment of π-calculus [9]. This allows the behaviour of communicating π-calculus processes to be specified with dependently-typed sessions and this additionally allows the typing of the terms communicated between processes. As an example, a process with the form $x(a).P$ is one that will receive input of a term a and then become the process P with all occurrences of a replaced by whatever was received. In contrast to linear types for λ-calculus where $A \multimap B$ was the type of a function, for π-calculus it types a service "offered" along

a channel like $x(a).P$ offers along x. Here, the input a must be a channel that "offers" a service of type A, then the process will become P which now instead offers service B along x.

The judgements for the session types have the form $\Psi, \Gamma, \Delta \vdash P :: x : A$ which reads as "Process P offers session type A along channel x using services Γ, Δ and under assumptions Ψ". The process P is a π calculus process, the session type A is a linear logic propositions, and x is the name of the channel along with the process's session is offered. The services in Δ of the form $x : A$ state that there is a session of type A offered along channel x that can be used once. The services in Γ also of the form $x : A$ offer replicating sessions of type A along x so can be used as many times as needed.

$$\frac{\Psi; \Gamma; \Delta_1 \vdash P :: x : A \quad \Psi; \Gamma; \Delta_2, x : A \vdash Q :: z : C}{\Psi; \Gamma; \Delta_1, \Delta_2 \vdash (\nu x)(P|Q) :: z : C}\text{cut}$$

Figure 4: Cut rule for Dependent Session Types

pi-calculus uses a sequence calculus for its formal proof system. The cut rule is shown in Figure 4 as an example construction rule. Here Q offers service type C along channel z but, as can be seen in the context, requires the use of a service A along the channel x. P offers service type A along channel x so these two processes can synchronise and be composed into process $(\nu x)(P|Q)$ which bind the channel x for the two processes to communicate along without external interference. The service type C is still offered along channel z while all communication through x happens internally.

2.3 π-calculus reduction with Dependent Session Types

Term reductions are also again linked to proof reductions for Linear types. In this case, each reduction is derived from the matching of a left and right rule with an application of cut, and then the reduction to instead applying cut directly to the initial formulas. An example of one of cut reductions for linear logic is shown in Figure 5. In this proof reduction, $\multimap R$ and $\multimap L$ are eliminated and we use two smaller cuts instead of one large one.

$$\frac{\dfrac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\multimap R \qquad \dfrac{\Delta_1 \vdash A \qquad \Delta_2, B \vdash C}{\Delta_1, \Delta_2, A \multimap B \vdash C}\multimap L}{\Delta, \Delta_1, \Delta_2 \vdash C}\text{cut}$$

$$\Longrightarrow$$

$$\dfrac{\dfrac{\Delta_1 \vdash A \qquad \Delta, A \vdash B}{\Delta, \Delta_1 \vdash B}\text{cut} \qquad \Delta_2, B \vdash C}{\Delta, \Delta_1, \Delta_2 \vdash C}\text{cut}$$

Figure 5: $\multimap$ cut reduction

If we now include the processes and channels from *pi*-calculus that correspond to the formulas as session types, we would then have the reduction in Figure 6. This then derives the reduction step of

$$\Delta, \Delta_1, \Delta_2 \vdash (\nu x)(x(y).P_1|(\nu w)x\langle w\rangle.(P_2|Q)) :: z : C$$

$$\Rightarrow$$

$$\Delta, \Delta_1, \Delta_2 \vdash (\nu x)((\nu w)(P_2|P_1)|Q) :: z : C$$

where a communication step has occurred along channel x between two processes in parallel.

$$\dfrac{\dfrac{\Delta, y : A \vdash P_1 :: x : B}{\Delta \vdash x(y).P_1 :: x : A \multimap B} \multimap R \quad \dfrac{\Delta_1 \vdash P_2 :: w : A \qquad \Delta_2, x : B \vdash Q :: z : C}{\Delta_1, \Delta_2, x : A \multimap B \vdash (\nu w)x\langle w\rangle.(P_2|Q) :: z : C} \multimap L}{\Delta, \Delta_1, \Delta_2 \vdash (\nu x)(x(y).P_1|(\nu w)x\langle w\rangle.(P_2|Q)) :: z : C} \text{cut}$$

$$\Longrightarrow$$

$$\dfrac{\dfrac{\Delta_1 \vdash P_2 :: w : A \qquad \Delta, y : A \vdash P_1 :: x : B}{\Delta, \Delta_1 \vdash (\nu w)(P_2|P_1) :: x : B} \text{cut} \qquad \Delta_2, x : B \vdash Q :: z : C}{\Delta, \Delta_1, \Delta_2 \vdash (\nu x)((\nu w)(P_2|P_1)|Q) :: z : C} \text{cut}$$

Figure 6: $\multimap$ cut reduction with session types

2.4 Term Passing for Dependent Session Types

Dependent Session Types deviate from the standard π-calculus in how they model type theory for not only channel communication but also communication of typed terms. These terms are functional terms where their type construction is not strictly defined but used assuming that some typed functional system exists. The assumptions in Ψ are the assumptions used for the typing of terms communicated within the process P. A basic hypothetical judgement form is given for terms $\Psi \vdash M : \tau$ where M is a term with type τ and Ψ is the assumed hypothesis. Judgements from Section 2.1 could for example be substituted here with Γ in place of Ψ, s in place of M and A in place of τ. The construction of communicated terms and their types is left arbitrary in order to facilitate the integration of any typed calculus into the system.

The addition of term passing also brings with it the addition of two new linear logic operators. These operators are $\forall x : \tau.A$ and $\exists y : \sigma.B$ where τ is a term type and A is a session type. $\forall$ indicates an input of a term and $\exists$ indicated a term output. These propositions are also what make the session types into dependent session types. If x occurs in A then the after an input to a session with type $\forall x : \tau.A$, The session will continue with type A but all occurrences of x in A will have been replaced. This means that A will be a different type "dependent" of the input x.

2.5 Example using Dependent Session Types

For this example the functional terms are assumed to be that of linear type theory from Section 2.1, and t is assumed to be a string. Take a process that offers a service of the following type

$$\forall \mathbf{f} : \text{Str} \to \text{Str}.\exists s : \text{Str}.\mathbf{1}$$

This type offers the simple service of inputting a function mapping strings to strings and then outputting a string. The construction for a judgement with this type is shown in Figure 7. The construction demonstrates how constructed terms can be integrated into the π-calculus in addition to the dependent session types.

$$\dfrac{\dfrac{}{\langle \mathbf{f} : Str \to Str\rangle \vdash \mathbf{f} : Str \to Str}\,\langle \mathrm{Id}\rangle \quad \dfrac{}{\langle t : Str\rangle \vdash t : Str}\,\langle \mathrm{Id}\rangle}{\langle \mathbf{f} : Str \to Str\rangle, \langle t : Str\rangle \vdash \mathbf{f} \cdot t : Str}\,\multimap -E$$

$$\dfrac{\dfrac{\langle \mathbf{f} : Str \to Str\rangle, \langle t : Str\rangle \vdash \mathbf{f} \cdot t : Str \qquad \dfrac{}{\langle \mathbf{f} : Str \to Str\rangle, \langle t : Str\rangle; \cdot; \cdot \Rightarrow \mathbf{0} :: x : \mathbf{1}}\,(\mathbf{1}R)}{\langle \mathbf{f} : Str \to Str\rangle, \langle t : Str\rangle; \cdot; \cdot \Rightarrow x\langle \mathbf{f} \cdot t\rangle.\mathbf{0} :: x : \exists s : \mathrm{Str}.\mathbf{1}}\,\exists R}{\langle t : Str\rangle; \cdot; \cdot \Rightarrow x(\mathbf{f}).x\langle \mathbf{f} \cdot t\rangle.\mathbf{0} :: x : \forall \mathbf{f} : \mathrm{Str} \to \mathrm{Str}.\exists s : \mathrm{Str}.\mathbf{1}}\,\forall R$$

Figure 7: Construction of judgement $\langle t : Str\rangle; \cdot; \cdot \Rightarrow x(\mathbf{f}).x\langle \mathbf{f} \cdot t\rangle.\mathbf{0} :: x : \forall \mathbf{f} : \mathrm{Str} \to \mathrm{Str}.\exists s : \mathrm{Str}.\mathbf{1}$

3 Audited Computation using Justification Logic

3.1 The JL$^{\natural}$ Fragment of Justification Logic

Justification Logic [10] is a logic where propositions can be paired with justifications that assert exactly how the proposition is known. These assertions are formed as $t : P$ which reads as "t is a justification that P is known". This is similar to how the Curry-Howard correspondence allows terms to be read as proof codes for propositions, and this similarity is utilised in [7] to internalise proof codes into propositions as justifications. Before covering audited computation, it defines a logic **JL**$^{\natural}$ which is a fragment of Justification Logic where propositions take the form $A ::= P | A \supset A | [\![\Sigma.s]\!]A$ where P is a propositional atom and $\Sigma.s$ is a proof code. The modality here is the same as the assertion $t : P$ where $[\![\Sigma.s]\!]A$ means $\Sigma.s$ is a justification for A. The syntax of a proof code s is shown in Figure 8

$$\begin{array}{rcll} s & ::= & a & \text{(Truth hypothesis)} \\ & | & \lambda a^{A}.s & \text{(Lambda function)} \\ & | & s \cdot s & \text{(Function application)} \\ & | & \langle u; \sigma\rangle & \text{(Renaming)} \\ & | & \Sigma.s & \text{(Modality introduction)} \\ & | & \mathrm{LET}(u^{A[\Sigma].s}, s) & \text{(Modality elimination)} \\ & | & \alpha\theta & \text{(Trail inspection)} \end{array}$$

Figure 8: Proof Code Syntax

This leads to another constructive logic system with intuitionistic judgements of the form $\Delta; \Gamma; \Sigma \vdash A | s$ which reads as "s is a justification that A is known under validity hypothesis Δ, truth hypotheses Φ, and trail hypothesis Σ". Here the proof codes bear witness to proofs of **JL**$^{\natural}$ propositions and encode each deduction made in the construction of a judgement. A validity hypothesis Δ contains variables of the form $u : A[\Sigma]$ which correspond to a propositional instance of a modality $[\![\Sigma.s]\!]A$. A truth hypothesis Φ contains variables of the form $a : A$ and correspond

to any typing judgement with type A. Finally, a trail hypothesis Σ contains variables of the form $\alpha : \mathbf{Eq}(A)$ which corresponds to a compatibility judgement for the type A.

These compatibility judgements take the form $\Delta; \Gamma; \Sigma \vdash \mathbf{Eq}(A, s, t)|e$ in which e is a compatibility code which asserts that s and t are compatible proof codes for the proposition A. e is made up of "trail constructors" which make up a proof of the compatibility of two proof codes s and t. The compatibility judgements serve two purposes. First they maintain the property of subject reduction for proof codes that should have the same type, and second they provide the construction of trails that will be used to track computation history for term reduction. The syntax of a compatibility code is given in Figure 9.

$$
\begin{array}{rcll}
e & ::= & \mathfrak{r}(s) & \text{(reflexivity)} \\
 & | & \mathfrak{s}(e) & \text{(symmetry)} \\
 & | & \mathfrak{t}(e, e) & \text{(transitivity)} \\
 & | & \mathfrak{ba}(a^A.s, s) & (\beta \text{ reduction}) \\
 & | & \mathfrak{bb}(u^{A[\Sigma]}.s, \Sigma.s) & (\beta_\Box \text{ reduction}) \\
 & | & \mathfrak{ti}(\theta, \alpha) & \text{(Trail inspection)} \\
 & | & \mathfrak{abs}(a^A.e) & \text{(Function compatibility)} \\
 & | & \mathfrak{app}(e, e) & \text{(Function application compatibility)} \\
 & | & \mathfrak{let}(u^{A[\Sigma]}.e, e) & \text{(Let compatibility)} \\
 & | & \mathfrak{trpl}(e_1, ..., e_{10}) & \text{(Replacement compatibility)}
\end{array}
$$

Figure 9: Compatibility Code Syntax

3.2 Terms of $\lambda^\mathfrak{h}$

In Section 2, we saw that terms were typed by propositions and terms were simultaneously proof codes. This is not the same for $\lambda^\mathfrak{h}$ typing judgements, where terms are typed by propositions, and this typing has a subsequent proof code rather than the terms being proof codes themselves. Judgements have the form $\Delta; \Gamma; \Sigma \vdash M : A|s$ which now reads as "s is evidence that M has type A under validity hypothesis Ψ, truth hypotheses Φ, and trail hypothesis Σ". The syntax of a $\lambda^\mathfrak{h}$ term is given in Figure 10.

$$
\begin{array}{rcll}
M & ::= & a & \text{(Term variable)} \\
& | & \lambda a^A.M & \text{(Lambda function)} \\
& | & MM & \text{(Function application)} \\
& | & \langle u;\sigma\rangle & \text{(Renaming)} \\
& | & !_e^\Sigma M & \text{(Audited term)} \\
& | & \text{LET } u^{A[\Sigma]} \text{ BE } M \text{ IN } N & \text{(Audited term composition)} \\
& | & \alpha\vartheta & \text{(Trail inspection)} \\
& | & e \rhd M & \text{(Term with derivation history)}
\end{array}
$$

Figure 10: $\lambda^\mathfrak{h}$ Term Syntax

The terms here are also the functional terms that we would like to record the computational history of. The key typing rule for this is the 'TBox' rule in Figure 11. Here we have a judgement for a term M with type A witnessed by proof code s. There is also a judgement for the compatibility of proof codes s and t for type a with compatibility code e. The rule creates what is referred to as an audited unit $!_e^\Sigma M$ which now internalises the compatibility code e which can also be seen as a record of the reduction trail for the term M. Corresponding to this is the internalisation of the proof code t into the propositional type along with binding the trail hypothesis Σ into the modality $[\![\Sigma.t]\!]A$.

$$
\frac{\Delta;\cdot;\Sigma \vdash M : A|s \quad \Delta;\cdot;\Sigma \vdash \mathbf{Eq}(A,s,t)|e}{\Delta;\Gamma;\Sigma' \vdash !_e^\Sigma M : [\![\Sigma.t]\!]A|\Sigma.t}\text{TBox} \qquad \frac{\alpha : \mathbf{Eq}(A) \in \Sigma \quad \Delta;\cdot;\cdot \vdash \vartheta : \mathcal{T}^B|\theta}{\Delta;\Gamma;\Sigma \vdash \alpha\vartheta : B|\alpha\theta}\text{TTI}
$$

Figure 11: TBox and TTI rules for $\lambda^\mathfrak{h}$ construction

There is also the need to query an audited unit's trail which is where trail replacement functions are used. Both ϑ and θ are referred to as trail replacements and they are functions on trail constructors that make up a compatability code e. ϑ maps trail constructors to terms and θ maps them to proof codes. We also have the type $\mathcal{T}^B$ which maps trail constructors to propositions that also depend on the type B. The trail replacement function defines how the trail should be interpreted which can be for example, a function that retrieves a set of permissions. This function is constructed in a judgement with a type and agreeing proof code. This judgement is then used to introduce a trail inspection in the 'TTI' rule in Figure 11 while also introducing a trail variable which represents the one time inspection of a trail for a given audited unit.

3.3 $\lambda^\mathfrak{h}$ Reduction

The behaviour of a functional system is defined by its reduction steps. A term that can no longer make any reduction steps is called a value V and a term that can take one or more steps is an

evaluation context $\mathcal{E}$. Principal reduction $\mapsto$ is defined by the rule

$$\frac{M \rightharpoonup N}{\mathcal{E}[M] \mapsto \mathcal{E}[M]}$$

which shows that any sub term M of $\mathcal{E}$ that can step by relation $\rightharpoonup$ to N can reduce within $\mathcal{E}$. The $\rightharpoonup$ relation is defined as the union between three reduction relations which represent the core computational steps taken by terms, shown in Figure 12. The first rule β^V is for function application, the second $\beta^V_\square$ is for substitution of a validity hypothesis, and the third $\mathcal{L}^V$ is for trail inspection within an audited unit. Each of these rules not only reduces the term but also introduces a trail constructor, for example $\mathfrak{ba}(a^A.s, s) \rhd M$, which is attached to the term to record the reduction step. Permutation reduction $\rightsquigarrow$ is then used to 'permute' this trail constructor out either to an audited unit operator and added into the compatibility code or to the front of the whole term.

$$\frac{\Delta;\Gamma_1, a{:}A;\Sigma_1 \vdash M{:}B \,|\, s \quad \Delta;\Gamma_2;\Sigma_2 \vdash V{:}A \,|\, t}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash (\lambda a^A.M)\ V \rightharpoonup \mathfrak{ba}(a^A.s,t) \rhd M^a_{V,t} : B \,|\, (\lambda a^A.s)\cdot t}\ \beta^V$$

$$\frac{\Delta;\cdot;\Sigma \vdash V{:}A \,|\, r \quad \Delta;\cdot;\Sigma \vdash \mathsf{Eq}(A,r,s) \,|\, e_1 \quad \Delta, u{:}A[\Sigma];\Gamma_2;\Sigma_2 \vdash N{:}C \,|\, t}{\Delta;\Gamma_{1,2};\Sigma_{1,2} \vdash O \rightharpoonup P : C^u_{\Sigma.s} \,|\, \mathrm{LET}(u^{A[\Sigma]}.t, \Sigma.s)}\ \beta^V_\square$$

$$\frac{\alpha{:}\mathsf{Eq}(A) \in \Sigma \quad \Delta';\cdot;\cdot \vdash \vartheta : \mathcal{T}^B \,|\, \theta \quad \Delta \subseteq \Delta' \quad \Delta;\cdot;\Sigma \vdash \mathcal{F}[\alpha\vartheta] : A \,|\, r \quad \Delta;\cdot;\Sigma \vdash \mathsf{Eq}(A,r,s) \,|\, e}{\Delta;\Gamma;\Sigma' \vdash !^{\Sigma}_{e}\mathcal{F}[\alpha\vartheta] \rightharpoonup !^{\Sigma}_{e}\mathcal{F}[\,\mathfrak{ti}(\theta,\alpha) \rhd e\vartheta\,] : [\![\,\Sigma.s\,]\!]A \,|\, \Sigma.s}\ \mathcal{I}^V$$

Figure 12: $\rightharpoonup$ relation

The result of these reductions is that all reduction steps that happen within audited units are recorded within the compatibility code of the audited unit operator. This is then used in the $\mathcal{L}^V$ reduction when the trail replacement function is applied to the compatibility code as it is when the reduction is applied. As stated earlier, this replacement function can be defined to retrieve whichever information is deemed relevant for the function being implemented.

4 Use of Audited Functional Terms in Dependently Typed Sessions

This auditing of reduction steps made by terms is only implemented for sequential λ-calculus functions. It is desirable, however, to have the ability to consult computation history in a concurrent setting as this is where a process may interact with an unknown source and require additional validity checks. The Dependent Session Types in Section 2.2 intentionally leave the definition of their functional layer of terms arbitrary. This potentially allows any formal functional typed language to be used by the system beyond just the simple linear types from Section 2.1. This suggests that $\lambda^{\mathfrak{h}}$ should be integrable into these dependent session types and then usable by the π-calculus processes to audit previous computation. This would not allow for auditing of another π-calculus process itself but would still prove useful for checking the validity of a term.

There is an inconsistency in the fact that the hypothetical judgement for dependent session types has the form $\Psi \vdash M : \tau$ whereas a typing for a $\lambda^{\mathfrak{h}}$ term is formed $\Delta; \Gamma; \Sigma \vdash M : A|s$. We extend the judgements of dependent session types to include the contexts contained in typing judgements of $\lambda^{\mathfrak{h}}$ term. We can also combine $\lambda^{\mathfrak{h}}$ terms M and $\mathbf{JL}^{\mathfrak{h}}$ proof codes s into pairs $(M|s)$ that can be recognised as a single term by dependent session types. This results in a new typing judgement for dependent session types of the form $\Psi; \Phi; \Sigma \vdash (M|s) : \tau$, where we have replaced the validity hypothesis label Δ with Ψ and the truth hypothesis label Γ with Φ to avoid conflict with the persistent and linear services of dependent session types, and M is a $\lambda^{\mathfrak{h}}$ of type τ as witnessed by proof code s.

We extend the judgements of dependent session types to include the contexts contained in Judgements of $\lambda^{\mathfrak{h}}$ terms. A judgement now takes the form $\Psi; \Phi; \Sigma; \Gamma; \Delta \vdash P :: x : A$. Here P is a process offering service of type A along channel x when composed with linear services Δ, persistent services Γ under validity hypothesis Ψ, truth hypotheses Φ, and trail hypothesis Σ. Most of the rules changes are trivial due to simply substituting in the new contexts in the place of Ψ and these trivial changes can be found in Figure 13.

$$\frac{}{\Psi; \Phi; \Sigma; \Gamma; x : A \Rightarrow [x \leftrightarrow z] :: z : A}\ \text{id} \qquad \frac{}{\Psi; \Phi; \Sigma; \Gamma; \cdot \Rightarrow \mathbf{0} :: x : \mathbf{1}}(\mathbf{1}R)$$

$$\frac{\Psi; \Phi; \Sigma; \Gamma; \Delta \Rightarrow P :: z : C}{\Psi; \Phi; \Sigma; \Gamma; \Delta, x : 1 \Rightarrow P :: z : C}(\mathbf{1}L) \qquad \frac{\Psi; \Phi; \Sigma; \Gamma; \cdot \Rightarrow P :: y : A}{\Psi; \Phi; \Sigma; \Gamma; \cdot \Rightarrow !z(y).P :: z : !A}!R$$

$$\frac{\Psi; \Phi; \Sigma; \Gamma, u : A; \Delta \Rightarrow P :: z : C}{\Psi; \Phi; \Sigma; \Gamma; \Delta, x : !A \Rightarrow P\{x \backslash u\} :: z : C}!L \qquad \frac{\Psi; \Phi; \Sigma; \Gamma, u : A; \Delta, y : A \Rightarrow P :: z : C}{\Psi; \Phi; \Sigma; \Gamma, u : A; \Delta \Rightarrow (\nu y)u\langle y\rangle.P :: z : C}\ \text{copy}$$

$$\frac{\Psi; \Phi; \Sigma; \Gamma; \Delta \Rightarrow P :: z : A \quad \Psi; \Phi; \Sigma; \Gamma; \Delta \Rightarrow Q :: z : B}{\Psi; \Phi; \Sigma; \Gamma; \Delta \Rightarrow z.\mathrm{case}(P, Q) :: z : A \& B}\&R$$

$$\frac{\Psi; \Phi; \Sigma; \Gamma; \Delta, x : A \Rightarrow P :: z : C}{\Psi; \Phi; \Sigma; \Gamma; \Delta, x : A \& B \Rightarrow x.\mathrm{inl}; P :: z : C}\&L_1 \qquad \frac{\Psi; \Phi; \Sigma; \Gamma; \Delta, x : B \Rightarrow P :: z : C}{\Psi; \Phi; \Sigma; \Gamma; \Delta, x : A \& B \Rightarrow x.\mathrm{inr}; P :: z : C}\&L_2$$

$$\frac{\Psi; \Phi; \Sigma; \Gamma; \Delta_1 \Rightarrow P :: y : A \quad \Psi; \Phi; \Sigma; \Gamma; \Delta_2 \Rightarrow Q :: z : B}{\Psi; \Phi; \Sigma; \Gamma; \Delta_1, \Delta_2 \Rightarrow (\nu y)z\langle y\rangle.(P|Q) :: z : A \otimes B}\otimes R \qquad \frac{\Psi; \Phi; \Sigma; \Gamma; \Delta, y : A, x : B \Rightarrow P :: z : C}{\Psi; \Phi; \Sigma; \Gamma; \Delta, x : A \otimes B \Rightarrow x(y).P :: z : C}\otimes L$$

$$\frac{\Psi; \Phi; \Sigma; \Gamma; \Delta \Rightarrow P :: z : A}{\Psi; \Phi; \Sigma; \Gamma; \Delta \Rightarrow z.\mathrm{inl}; P :: z : A \oplus B} \oplus R_1 \qquad \frac{\Psi; \Phi; \Sigma; \Gamma; \Delta \Rightarrow P :: z : B}{\Psi; \Phi; \Sigma; \Gamma; \Delta \Rightarrow z.\mathrm{inr}; P :: z : A \oplus B} \oplus R_2$$

$$\frac{\Psi; \Phi; \Sigma; \Gamma; \Delta, x : A \Rightarrow P :: z : C \quad \Psi; \Phi; \Sigma; \Gamma; \Delta, x : B \Rightarrow Q :: z : C}{\Psi; \Phi; \Sigma; \Gamma; \Delta, x : A \oplus B \Rightarrow x.case(P, Q) :: z : C} \oplus L$$

$$\frac{\Psi; \Phi; \Sigma; \Gamma; \Delta_1 \Rightarrow P :: x : A \quad \Psi; \Phi; \Sigma; \Gamma; \Delta_2, x : A \Rightarrow Q :: z : C}{\Psi; \Phi; \Sigma; \Gamma; \Delta_1, \Delta_2 \Rightarrow (\nu x)(P|Q) :: z : C}\text{cut}$$

$$\frac{\Psi; \Phi; \Sigma; \Gamma; \cdot \Rightarrow P :: x : A \quad \Psi; \Phi; \Sigma; \Gamma, u : A; \Delta \Rightarrow Q :: z : C}{\Psi; \Phi; \Sigma; \Gamma; \Delta \Rightarrow (\nu x)((!u(x).P)|Q) :: z : C}\text{cut!}$$

Figure 13: Rules and Axioms for Dependent Session Types with $\lambda^{\mathfrak{h}}$

The non trivial rules are those that use the typed functional terms. For the rules $\forall L$ and $\exists R$ a

change needs to be made in the substitution as the terms now consist of a pair of variables instead of individual. Changing the substitution to rename the parts of the term pairs separately allows them, term and proof code, to possible be split and used separately without losing the validity of the rules. The rules are then as follows:

$$\frac{\Psi;\Phi;\Sigma \vdash (N|s) : \tau \quad \Psi;\Phi;\Sigma;\Gamma;\Delta, x : A\{N/y, s/t\} \Rightarrow P :: z : C}{\Psi;\Phi;\Sigma;\Gamma;\Delta, x : \forall(y|t) : \tau.A \Rightarrow x\langle(N|s)\rangle.P :: z : C}\forall L$$

$$\frac{\Psi;\Phi;\Sigma \vdash (N|s) : \tau \quad \Psi;\Phi;\Sigma;\Gamma;\Delta \Rightarrow P : A\{N/x, s/t\}}{\Psi;\Phi;\Sigma;\Gamma;\Delta \Rightarrow z\langle(N|s)\rangle.P :: z : \exists(x|t) : \tau.A}\exists R$$

The last two rules $\forall R$ and $\forall L$ which access the hypothesis Ψ for a given typing. For dependent session types this typing is assumed to be intuitionistic and not a linear judgement which is only the case for hypotheses in the validity hypothesis Δ of $\lambda^{\mathfrak{h}}$ typing judgements. This means that these rules will only be applicable for a type of the form $[\![\Sigma.s]\!]A$ and for audited unit terms. This restriction is however not a disadvantage as it insures that terms passed will always be of an audited form and so it can be assumed their trail can be inspected in some way without any manipulation of the term. The rules are then as follows:

$$\frac{\Psi, x : \tau[\Sigma_1];\Phi;\Sigma_2;\Gamma;\Delta \Rightarrow P :: z : A}{\Psi;\Phi;\Sigma_2;\Gamma;\Delta \Rightarrow z(x)P :: z : \forall x : \tau.A}\forall R \qquad \frac{\Psi, y : \tau[\Sigma_1];\Phi;\Sigma_2;\Gamma;\Delta, x : A \Rightarrow P :: z : C}{\Psi;\Phi;\Sigma_2;\Gamma;\Delta, x : \exists y : \tau.A \Rightarrow x(y).P :: z : C}\exists L$$

The π-calculus and their typing scheme would need to be expanded further for them to be able to directly audit functional terms and control reduction steps based on this. However, it is likely that π-calculus processes can simply audit a functional term using another, though it is not clear yet how this can be done. The problem here arises in how *pi*-calculus can use the trail inspection functions to perform some alternate behaviour depending on the history of audited terms. The trail inspection operator is internal to $\lambda^{\mathfrak{h}}$ and trail variables that they use are bound with audited terms.

With this new typing scheme dependent session types can also model services where functional terms are communicated along with their computational history. One such example using this could be a server which offers the service to input a functional program and have it executed on the server. This server may then require certain permissions which are held by the given program after first executing some verification checks.

5 Conclusion and Future Work

The topics of propositional type theory, dependent session types and audited functional terms have been investigated and compiled into a high level summary. This has then supported a speculative summary of how these areas of research could be combined to facilitate the expression of concurrent processes that can pass terms that record their reduction history. We have been able to extend the typing scheme of dependent session types for the passing of $\lambda^{\mathfrak{h}}$ terms. It has been identified exactly how the history of $\lambda^{\mathfrak{h}}$ terms are recorded and additionally how they are inspected during computation.

However, the speculation still leaves several steps to be investigated. As a follow on from this project, it could be specified in more depth how to externally audit a $\lambda^{\mathfrak{h}}$ term using another functional term of the same form. π-calculus and dependent session types could also be extended

to be able to also directly inspect the trails of functional terms and control their reduction based on this. This would require the formulation of additional rules, reductions, and expansions. One theory is for *pi*-calculus to be extended to have audited inputs that bind some set of trail variables corresponding to the term that will be input, and this input step can only take place after the application of some trail inspection functions for each trail variable.

Finally, concurrent processes could also record their own reduction steps though this may require a significant restructure of their typing schemes as $\lambda^{\mathfrak{h}}$ terms use a comparatively very different form of logic for this purpose.

6 Personal Evaluation

The initial goal of this project was to investigate the possibility of the combination of two areas of research connected by a similar approach to computational theory and Curry-Howard isomorphisms. The two areas were, however, fairly distinct in the area of logic that they come from. When I started this project I was unfamiliar with both the area of linear logic and justification logic. So before investigating the combination of the two systems a large proportion of the project was spent building up a background knowledge of each line of research and the steps that led to each result.

In the time I have made significant progress in understanding linear typed, dependent session types, and audited functional terms with justification logic. This has, however, resulted in less concrete formulation and results. For example, while some initial construction rules have been formulated, a working example that really demonstrates the potential expression of this new system has not been realised.

I hope that the summary of each area, from the overarching perspective of type theory in particular, will help in further developing the ideas and speculation that has been presented in Section 4.

References

[1] Maribel Fernández. *Models of computation: an introduction to computability theory*. Springer Science & Business Media, 2009.

[2] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[3] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[4] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[5] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, nov 2015.

[6] Cédric Fournet. Access control based on execution history. In *10th Annual Network and Distributed System Security Symposium (NDSS'03)*, February 2003.

[7] Francisco Bavera and Eduardo Bonelli. Justification logic and audited computation. *Journal of Logic and Computation*, 28:909–934, 07 2018.

[8] Philip Wadler. A taste of linear logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, MFCS '93, page 185–210, Berlin, Heidelberg, 1993. Springer-Verlag.

[9] Luís Caires, Frank Pfenning, and Bernardo Toninho. Towards concurrent type theory. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Sergei Artemov. The logic of justification. *The Review of Symbolic Logic*, 1:477 – 513, 12 2008.