Thesis Proposal: Session-Typed Concurrent Contracts

Hannah Gommerstadt

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning (chair)
Limin Jia
Jan Hoffman
Bernardo Toninho (Universidade NOVA de Lisboa)
Adrian Francalanza (University of Malta)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: contracts, session types, monitoring

Abstract

Multi-process systems control the behavior of everything from datacenters storing our information to banking systems managing money. Each one of these processes has a prescribed role, their contract, that governs their behavior during the joint computation. When a single process violates their communication contract, the impact of this misbehavior can rapidly propagate through the system. This thesis develops techniques for dynamically monitoring expressive classes of concurrent contracts. We provide multiple mechanisms to monitor contracts of increasing complexity. In order to model message-passing concurrent computation, we use a session type system. First, we present a method for dynamic monitoring and blame assignment where communication contracts are expressed using session types. Second, we describe contract-checking processes that handle stateful contracts that cannot be expressed with a session type. These contract-checking processes are also able to encode type refinements. In the proposed work, we aim to encode dependent types in our system which will allow us to monitor complex invariants. Finally, we propose a number of other monitoring extensions including a formal mechanism to analyze the resource consumption of our monitoring processes.

Contents

List of Figures

Chapter 1

Introduction

Multi-process systems control the behavior of everything from datacenters storing our information to banking systems managing money. Each one of these processes has a prescribed role, their *contract*, that governs their behavior during the joint computation. When a single process violates their communication contract, the impact of this misbehavior can rapidly propagate through the system. This thesis develops techniques to detect and contain this misbehavior by dynamically monitoring violations of a process' contract.

In functional languages, a contract for a function can be modeled as an expressive type that places constraints on its arguments and return value [14]. Typically, as the function executes, these constraints are checked dynamically. If a constraint is violated, the system will attempt to assign blame to the party responsible for the contract violation. While significant research has been done on dynamic contract checking in a sequential setting, concurrent computation presents an additional challenge.

In order to model concurrent computation, we use a session type system which was designed by Toninho et al. [36]. Session types are based on a computational interpretation of linear logic which is a substructural logic that allows reasoning about resources within the logic itself. In this logic, all resources are ephemeral and are consumed with each step of the computation. That is, we annotate each communication channel with a session type and as processes communicate over the channel by message passing, the type of the channel changes with every step of the communication.

We can think of each channel's session type as prescribing the communication contract on that channel – for example, a session type could require that first two integers, x and y, be sent over a channel, and then an integer response z be received. As each integer is sent and received, the type of the channel is updated to reflect the current communication contract. However, a malicious process could take over and send some nefarious string over this channel. This is clearly a violation of the communication contract which should cause a system-wide alarm.

Thesis Statement: Session typed monitors give rise to novel techniques for dynamically monitoring expressive classes of concurrent contracts and provide strong theoretical guarantees (safety and transparency).

Our first contribution is to dynamically monitor that each channel's communication contract

is being upheld. If a suspicious string is observed flowing through this channel, our monitors will detect that the message is inconsistent with the channel's type and raise an alarm. We have designed a mechanism where monitors are placed on the communication channels and act as type-checkers to ascertain whether communication on a channel is consistent with its contract. If the messages follow the protocol, the monitor lets them through with no observable change to the computation. If the messages disobey the protocol, the monitor raises an alarm and blames the process or processes responsible for the faulty messages.

We now consider a more complicated contract which would ensure that the integer response z is the sum of the integers x and y. Monitoring this contract requires designing a system where monitors can maintain internal state. In this case, the monitor needs to store the values of x and y in order to compute their sum and compare it with the integer z. To handle this class of contracts, we have developed partial identity monitors. These monitors are able to maintain state and perform calculations which allows them to check complicated properties such as whether a list of parenthesis is matched or a list is in sorted order. The partial identity processes execute concurrently with the process being monitored. They are observably equivalent to the identity process, up to termination, thus ensuring transparency of our monitors. Further, our partial identity monitors are also able to encode refinement contracts. That is, we can generate a monitor to check whether a refinement from one type to another type causes an error.

The goal of this thesis is to extend the monitoring frontier to be able to express more classes of contracts with session-typed monitors. In the remaining work, we aim to encode dependent types in our system which will allow us to monitor many complex invariants. Monitoring dependent contracts is challenging because usual encodings of dependent types involve sending proof objects through the system. These proof objects must be generated and transmitted through the system, which requires significant infrastructure [26]. We aim to provide a more lightweight approach by generating partial identity monitors for dependent session types containing on proof objects. These monitors will then check the constraints encoded in the proof objects at runtime. Finally, we propose a number of other monitoring extensions including a formal mechanism to analyze the resource consumption of our monitoring processes.

The rest of this proposal is organized as follows. Chapter 2 provides background on session types and contracts. Chapter 3 presents results on dynamic monitoring in an untrusted setting. Chapter 4 introduces partial identity monitors and examines key examples. Chapter 5 and Chapter 6 explore the encoding of refinement and dependent types, respectively, into partial identity monitors. Chapter 7 reviews proposed miscellaneous monitoring extensions. Finally, Chapter 8 outlines a timeline for the thesis.

Chapter 2

Background

In this chapter we first provide background for using session types to reason about concurrent computation. We then provide examples of contract checking in a functional language and survey recent work.

2.1 Session Types

Session types prescribe the communication behavior of message-passing concurrent processes. We approach them here via their foundation in intuitionistic linear logic [4, 5, 34]. The key idea is that an intuitionistic linear sequent

$$A_1, \ldots, A_n \vdash C$$

is interpreted as the interface to a *process expression* P. We label each of the antecedents with a channel name a_i and the succedent with a channel name c. The a_i are the channels *used* and c is the channel *provided* by P.

$$a_1 : A_1, \ldots, a_n : A_n \vdash P :: (c : C)$$

We abbreviate the antecedents by Δ. All the channels a_i and c must be distinct, and bound variables may be silently renamed to preserve this invariant in the rules. Furthermore, the antecedents are considered modulo exchange. Cut corresponds to parallel composition of two processes that communicate along a private channel x, where P is the *provider* along x and Q the *client*.

$$\frac{\Delta \vdash P :: (x : A) \quad x : A, \Delta' \vdash Q :: (c : C)}{\Delta, \Delta' \vdash x{:}A \leftarrow P \; ; \; Q :: (c : C)} \; \mathsf{cut}$$

Operationally, the process $x : A \leftarrow P \; ; \; Q$ spawns P as a new process and continues as Q, where P and Q communicate along a fresh channel a, which is substituted for x. We sometimes omit the type A of x in the syntax when it is not relevant or can be inferred.

In order to define the operational semantics rigorously, we use *multiset rewriting* [6]. The configuration of executing processes is described as a collection $\mathcal{C}$ of propositions $\mathsf{proc}(c, P)$

(process P is executing, providing along c) and $\mathsf{msg}(c, M)$ (message M is sent along c). All the channels c provided by processes and messages in a configuration must be distinct.

To begin with, a cut just spawns a new process, and is in fact the only way new processes are spawned. We describe a transition $\mathcal{C} \longrightarrow \mathcal{C}'$ by defining how a subset of $\mathcal{C}$ can be rewritten to a subset of $\mathcal{C}'$, possibly with a freshness condition that applies to all of $\mathcal{C}$ in order to guarantee the uniqueness of each channel provided.

$$\mathsf{proc}(c, x{:}A \leftarrow P\ ;\ Q) \longrightarrow \mathsf{proc}(a, [a/x]P), \mathsf{proc}(c, [a/x]Q) \quad \textit{(a fresh)}$$

Each of the connectives of linear logic then describes a particular kind of communication behavior which we capture in similar rules. Before we move on to that, we consider the identity rule, in logical form and operationally.

$$\frac{}{A \vdash A}\ \mathsf{id} \qquad \frac{}{b : A \vdash a \leftarrow b :: (a : A)}\ \mathsf{id} \qquad \mathsf{proc}(a, a \leftarrow b), \mathcal{C} \longrightarrow [b/a]\mathcal{C}$$

Operationally, it corresponds to identifying the channels a and b, which we implement by substituting b for a in the remainder $\mathcal{C}$ of the configuration (which we make explicit in this rule). The process offering a terminates. We refer to $a \leftarrow b$ as *forwarding* since any messages along a are instead "forwarded" to b.

We consider each class of session type constructors, describing their process expression, typing, and asynchronous operational semantics. The linear logical semantics can be recovered by ignoring the process expressions and channels.

Internal and external choice Even though we distinguish a *provider* and its *client*, this distinction is orthogonal to the direction of communication: both may either send or receive along a common private channel. Session typing guarantees that both sides will always agree on the direction and kind of message that is sent or received, so our situation corresponds to so-called *binary session types* [20].

First, the *internal choice* $c : A \oplus B$ requires the provider to send a token inl or inr along c and continue as prescribed by type A or B, respectively. For convenience, we support n-ary labelled choice $\oplus\{\ell : A_\ell\}_{\ell \in L}$ where L is a set of labels. A process providing $c : \oplus\{\ell : A_\ell\}_{\ell \in L}$ sends a label $k \in L$ along c and continues with type A_k. The client will operate dually, branching on a label received along c.

$$\frac{k \in L \quad \Delta \vdash P :: (c : A_k)}{\Delta \vdash c.k\ ;\ P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R \qquad \frac{\Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)}\ \oplus L$$

The operational semantics is somewhat tricky, because we communicate asynchronously. We need to spawn a message carrying the label ℓ, but we also need to make sure that the *next* message send along the same channel does not overtake the first (which would violate session fidelity). Sending a message therefore creates a fresh continuation channel c' for further communication, which we substitute in the continuation of the process. Moreover, the recipient also switches to this continuation channel after the message is received.

$$\mathsf{proc}(c, c.k\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, c.k\ ;\ c \leftarrow c') \quad \textit{(c' fresh)}$$
$$\mathsf{msg}(c, c.k\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \longrightarrow \mathsf{proc}(d, [c'/c]Q_k)$$

It is interesting that the message along c, followed by its continuation c' can be expressed as a well-typed process expression using forwarding $c.k\ ;\ c \leftarrow c'$. This pattern will work for all other pairs of send/receive operations.

External choice reverses the roles of client and provider, both in the typing and the operational rules. Below are the semantics and the typing is in Fig. 5.7.

$$\begin{array}{l}\mathsf{proc}(d, c.k\ ;\ Q) \longrightarrow \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh}) \\ \mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c]P_k)\end{array}$$

Sending and receiving channels Session types are *higher-order* in the sense that we can send and receive channels along channels. Sending a channel is perhaps less intuitive from the logical point of view, so we show that and just summarize the rules for receiving.

If we provide $c : A \otimes B$, we send a channel $a : A$ along c and continue as B. From the typing perspective, it is a restricted form of the usual two-premise $\otimes R$ rule by requiring the first premise to be an identity. This restriction separates spawning of new processes from the sending of channels.

$$\frac{\Delta \vdash P :: B}{\Delta, a : A \vdash \mathsf{send}\ c\ a\ ;\ P :: (c : A \otimes B)}\ \otimes R^* \qquad \frac{\Delta, x : A, c : B \vdash Q :: (d : D)}{\Delta, c : A \otimes B \vdash x \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \otimes L$$

The operational rules follow the same patterns as the previous case.

$$\begin{array}{l}\mathsf{proc}(c, \mathsf{send}\ c\ a\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ a\ ;\ c \leftarrow c') \quad \textit{(c' fresh)} \\ \mathsf{msg}(c, \mathsf{send}\ c\ a\ ;\ c \leftarrow c'), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c][a/x]Q)\end{array}$$

Receiving a channel (written as a linear implication $A \multimap B$) works symmetrically. Below are the semantics and the typing is shown in Figure 5.7.

$$\begin{array}{l}\mathsf{proc}(d, \mathsf{send}\ c\ a\ ;\ Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ a\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh}) \\ \mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ a\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c][a/x]P)\end{array}$$

Termination We have already seen that a process can terminate by forwarding. Communication along a channel ends explicitly when it has type $\mathbf{1}$ (the unit of $\otimes$) and is closed. By linearity there must be no antecedents in the right rule.

$$\frac{}{\cdot \vdash \mathsf{close}\ c :: (c : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Delta \vdash Q :: (d : D)}{\Delta, c : \mathbf{1} \vdash \mathsf{wait}\ c\ ;\ Q :: (d : D)}\ \mathbf{1}L$$

Since there cannot be any continuation, the message takes a simple form.

$$\begin{array}{l}\mathsf{proc}(c, \mathsf{close}\ c) \longrightarrow \mathsf{msg}(c, \mathsf{close}\ c) \\ \mathsf{msg}(c, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, Q)\end{array}$$

Quantification First-order quantification over elements of domains such as integers, strings, or booleans allows ordinary basic data values to be sent and received. At the moment, since we

have no type families indexed by values, the quantified variables cannot actually appear in their scope. This will change in Section 5 so we anticipate this in these rules.

In order to track variables ranging over values, a new context Ψ is added to all judgments and the preceding rules are modified accordingly. All value variables n declared in Ψ must be distinct. Such variables are not linear, but can be arbitrarily reused, and are therefore propagated to all premises in all rules. We write $\Psi \vdash v : \tau$ to check that value v has type τ in context Ψ.

$$\frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta \vdash P :: (c : [v/n]A)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ v\ ;\ P :: (c : \exists n{:}\tau.\ A)}\ \exists R \qquad \frac{\Psi, n{:}\tau\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \exists n{:}\tau.\ A \vdash n \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \exists L$$

$$\begin{array}{l}\mathsf{proc}(c, \mathsf{send}\ c\ v\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ v\ ;\ c \leftarrow c') \quad (c' \text{ fresh}) \\ \mathsf{msg}(c, \mathsf{send}\ c\ v\ ;\ c \leftarrow c'), \mathsf{proc}(d, n \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c][v/n]Q)\end{array}$$

The situation for universal quantification is symmetric. The semantics are given below and the typing is shown in Figure 5.7.

$$\begin{array}{l}\mathsf{proc}(d, \mathsf{send}\ c\ v\ ;\ Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ v\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh}) \\ \mathsf{proc}(c, n \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ v\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c][v/n]P)\end{array}$$

Processes may also make internal transitions while computing ordinary values, which we don't fully specify here. Such a transition would have the form

$$\mathsf{proc}(c, P[e]) \longrightarrow \mathsf{proc}(c, P[e']) \quad \text{if} \quad e \mapsto e'$$

where $P[e]$ would denote a process with an ordinary value expression in evaluation position and $e \mapsto e'$ would represent a step of computation.

Shifts Finally, we come to shifts. For the purpose of monitoring, it is important to track the direction of communication. To make this explicit, we *polarize* the syntax and use so-called *shifts* to change the direction of communication. For more detail, see Pfenning and Griffith [29].

$$\begin{array}{lllll} \text{Negative types} & A^-, B^- & ::= & \&\{\ell : A^-_\ell\}_{\ell \in L} \mid A^+ \multimap B^- \mid \forall n{:}\tau.\ A^- \mid \uparrow A^+ \\ \text{Positive types} & A^+, B^+ & ::= & \oplus\{\ell : A^+_\ell\}_{\ell \in L} \mid A^+ \otimes B^+ \mid 1 \mid \exists n{:}\tau.\ A^+ \mid \downarrow A^- \\ \text{Types} & A, B, C, D & ::= & A^- \mid A^+ \end{array}$$

From the perspective of the provider, all negative types receive and all positive types send. It is then clear that $\uparrow A$ must receive a shift message and then start sending, while $\downarrow A$ must send a shift message and then start receiving. For this restricted form of shift, the logical rules are otherwise uninformative. The semantics are given below and the typing is shown in Figure 5.7.

$$\begin{array}{l}\mathsf{proc}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c \leftarrow c') \quad (c' \text{ fresh}) \\ \mathsf{msg}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c]Q) \\ \\ \mathsf{proc}(d, \mathsf{send}\ c\ \mathsf{shift}\ ;\ Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ \mathsf{shift}\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh}) \\ \mathsf{proc}(c, \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ \mathsf{shift}\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c]P)\end{array}$$

Recursive types Practical programming with session types requires them to be recursive, and processes using them also must allow recursion. For example, lists with elements of type int can be defined as the purely positive type list^+.

$$\mathsf{list}^+ = \oplus\{\mathsf{cons} : \exists n{:}\mathsf{int}.\mathsf{list}^+ \ ; \ \mathsf{nil} : \mathbf{1}\}$$

A provider of type $c : \mathsf{list}$ is required to send a sequence such as $\mathsf{cons} \cdot v_1 \cdot \mathsf{cons} \cdot v_2 \cdots$ where each v_i is an integer. If it is finite, it must be terminated with $\mathsf{nil} \cdot \mathsf{end}$ where the end message is shorthand for $\mathsf{msg}(c, \mathsf{close}\ c)$. We show all the forms a single message could take in Figure **??**. In the form of a grammer, we could write

$$\mathit{From} ::= \mathsf{cons} \cdot v \cdot \mathit{From} \mid \mathsf{nil} \cdot \mathsf{end}$$

A second example is a multiset (bag) of integers, where the interface allows inserting and removing elements, and testing if it is empty. If the bag is empty when tested, the provider terminates after responding with the empty label.

$$\begin{array}{rcl} \mathsf{bag}^- & = & \&\{\mathsf{insert} : \forall n{:}\mathsf{int}.\mathsf{bag}^- \ ; \\ & & \mathsf{remove} : \forall n{:}\mathsf{int}.\mathsf{bag}^- \ ; \\ & & \mathsf{is\_empty} : \uparrow \oplus\{\mathsf{empty} : \mathbf{1} \ ; \ \mathsf{nonempty} : \downarrow \mathsf{bag}^-\}\} \end{array}$$

The protocol now describes the following grammar of exchanged messages, where *To* goes to the provider, *From* comes from the provider, and v stands for integers.

$$\begin{array}{lcl} \mathit{To} & ::= & \mathsf{insert} \cdot v \cdot \mathit{To} \mid \mathsf{remove} \cdot v \cdot \mathit{To} \mid \mathsf{is\_empty} \cdot \mathsf{shift} \cdot \mathit{From} \\ \mathit{From} & ::= & \mathsf{empty} \cdot \mathsf{end} \mid \mathsf{nonempty} \cdot \mathsf{shift} \cdot \mathit{To} \end{array}$$

For these protocols to be realized in this form and support rich subtyping and refinement types without change of protocol, it is convenient for recursive types to be *equirecursive*. This means a defined type such as list^+ is viewed as *equal* to its definition $\oplus\{\ldots\}$ rather than *isomorphic*. For this view to be consistent, we require type definitions to be *contractive* [15], that is, they need to provide at least one send or receive interaction before recursing.

The most popular formalization of equirecursive types is to introduce an explicit μ-constructor. For example,

$$\mathsf{list} = \mu\alpha.\, \oplus\{\ \mathsf{cons} : \exists n{:}\mathsf{int}.\, \alpha, \mathsf{nil} : \mathbf{1}\ \}$$

with rules unrolling the type $\mu\alpha.\, A$ to $[(\mu\alpha.\, A)/\alpha]A$. An alternative (see, for example, Balzers and Pfenning [2]) is to use an explicit definition just as we stated, for example, list and bag, and consider the left-hand side *equal* to the right-hand side in our discourse. In typing, this works without a hitch. When we consider subtyping explicitly, we need to make sure we view inference systems on types as being defined *co-inductively*. Since a co-inductively defined judgment essentially expresses the absence of a counterexample, this is exactly what we need for the operational properties like progress, preservation, or absence of blame. We therefore adopt this view.

Recursive processes In addition to recursively defined types, we also need recursively defined processes. We follow the general approach of Toninho et al. [36] for the integration of a (functional) data layer into session-typed communication. A process can be named p, ascribed a type, and be defined as follows.

$$\begin{array}{l} p : \forall n_1{:}\tau_1.\ \ldots\ \forall n_k{:}\tau_k.\{A \leftarrow A_1, \ldots, A_m\} \\ x \leftarrow p\, n_1\ \ldots\ n_k \leftarrow y_1, \ldots, y_m = P \end{array}$$

where we check $(n_1{:}\tau_1, \ldots, n_k{:}\tau_k)\ ;\ (y_1{:}A_1, \ldots, y_m{:}A_m) \vdash P :: (x : A)$

We use such process definitions when spawning a new process with the syntax

$$c \leftarrow p\, e_1\ \ldots, e_k \leftarrow d_1, \ldots, d_m\ ;\ Q$$

which we check with the rule

$$\frac{(\Psi \vdash e_i : \tau_i)_{i\in\{1,\ldots,k\}} \quad \Theta = [e_1/n_1, \ldots, e_k/n_k] \quad \Delta' = [\Theta](d_1{:}A_1, \ldots, d_m{:}A_m) \quad \Psi\ ;\ \Delta, c : [\Theta]A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, \Delta' \vdash c \leftarrow p\, e_1 \ldots e_k \leftarrow d_1, \ldots, d_m\ ;\ Q :: (d : D)}\ \mathsf{pdef}$$

After evaluating the value arguments, the call consumes the channels d_j (which will not be available to the continuation Q, due to linearity). The continuation Q will then be the (sole) client of c and The new process providing c will execute $[c/x][d_1/y_1] \ldots [d_m/y_m]P$.

One more quick shorthand used in the examples: a tail-call $c \leftarrow p\ \overline{e} \leftarrow \overline{d}$ in the definition of a process that provides along c is expanded into $c' \leftarrow p\ \overline{e} \leftarrow \overline{d}\ ;\ c \leftarrow c'$ for a fresh c'. Depending on how forwarding is implemented, however, it may be much more efficient [19].

Stopping computation Finally, in order to successfully monitor computation, we need the capability to assert conditions and stop the computation at particular points. We add assert blocks to check conditions on observable values and an abort block to stop computation. We tag the assert and abort blocks with a label l which allows us to determine which assertion failed when the computation aborts. The semantics are given below and the typing is in Figure 5.7.

$$\mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{True}; Q) \longrightarrow \mathsf{proc}(c, Q) \qquad \mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{False}; Q) \longrightarrow \mathsf{abort}(l)$$

We overload the abort(l) notation to refer to both the semantic construct (as shown above) and the process expression (used frequently in our examples in Section 3). Progress and preservation were proven for the above system, with the exception of the abort and assert rules, in prior work [29]. The additional proof cases do not change the proof significantly. We summarize the process expressions in Figure 2.1.

2.2 Contracts

A contract for a function can be modeled by an expressive type that places constraints on the arguments and return value. For example, consider a function f where both the argument and the return value must be positive. We first write the standard type:

$$f : \mathsf{int} \to \mathsf{int}$$

We can now define a more precise type $\mathsf{posInt} = \{x : \mathsf{int} \mid x > 0\}$. This type is a refinement of the integer type and can be used to express the desired contract.

$$f : \mathsf{posInt} \to \mathsf{posInt}$$

If an argument to f is not positive, then f's caller is blamed for the contract violation. Symmetrically, if f's result is not positive, the blame falls on f itself. Unfortunately, this simple

$P, Q, R ::=$	
$\mathsf{close}\ c$	send end and terminate
$\mathsf{wait}\ c\ ;\ Q$	recv end, continue with Q
$\mathsf{send}\ c\ a\ ;\ Q$	send channel a along c, and continue as Q
$x \leftarrow \mathsf{recv}\ c\ ;\ Q_x$	receive a along c, continue as Q_a
$c.\ell_j\ ;\ Q$	send ℓ_j along c, continue as Q
$\mathsf{case}\ c\ \mathsf{of}\ \{\ell_i \Rightarrow Q_i\}_i$	recv ℓ_j along c, cont. as Q_j
$\mathsf{send}\ c\ v\ ;\ Q$	send value v along c, continue as Q
$n \leftarrow \mathsf{recv}\ c\ ;\ Q_n$	recv value v along c, continue as Q_v
$\mathsf{send}\ c\ \mathsf{shift}\ ;\ Q$	send shift along c, continue as Q
$\mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q$	receive shift along c, continue as Q
$x \leftarrow P_x\ ;\ Q_x$	create new a, spawn P_a, continue as Q_a
$c \leftarrow d$	connect c with d and terminate
$\mathsf{assert}\ l\ p\ ;\ Q$	assert predicate p with label l and continue as Q
$\mathsf{abort}\ l$	abort with label l

Figure 2.1: Process expressions

approach to contract checking fails to generalize to a higher-order setting. Consider the below function:

$$g : (\mathsf{posInt} \to \mathsf{posInt}) \to \mathsf{posInt}$$

The contract's domain accepts functions that map positive integers to positive integers. The contract's range obliges g to produce positive numbers. The function g could be passed a function $f : \mathsf{posInt} \to \mathsf{posInt}$ which matches g's domain or a function that has a stricter domain such as $f : \mathsf{posInt} \to \{x : \mathsf{posInt} \mid x > 10\}$. The key insight here is that a contract checker cannot determine if g's argument meets its contract when g is called. It must wait until this argument, say the function f, is applied to another argument to validate its contract. This notion of deferred contract checking for higher-order functions, first introduced by Findler and Felleisen [14], allows the contract checker to assign blame to the party that actually violated the contract. When performing contract-checking for session-types in a higher-order setting (described in Section 3), we use a similar approach.

Contracts are frequently used to prescribe the interactions between code typed with different levels of precision. In an extreme case, contracts can be used to integrate code that is typed with code that is untyped to ensure that dynamically-typed code maintains statically-typed invariants. In this dissertation, we frequently use contracts to connect session types and refinements of those types.

Wadler and Findler [38] define a type system with casts, called a blame calculus, where casts represent contracts. In their system, a contract is modeled as a cast from a source type S to a target type T with a blame label p. The source term s has type S while the whole term has type T.

$$\langle T \Leftarrow S \rangle^p s$$

In this situation, blame is assigned to the label p when the term contained in the cast, s in this example, fails to satisfy the contract associated with the cast. Conversely, the complement of p, written $\overline{p}$ is blamed, when the context containing the cast fails to satisfy the contract. A cast will be dynamically checked to validate whether a given value can be coerced to the required type.

Consider the following cast which takes a function with a domain and range of type int to a more precise domain and range.

$$(\langle \mathsf{posInt} \to \mathsf{posInt} \Leftarrow \mathsf{int} \to \mathsf{int} \rangle^{p_1} f) x$$

When this cast is checked, it will be broken into two casts, one for the range and one for the domain. The cast for the range of the function will attempt to cast the range of the source to the range of the target as follows: $\langle \mathsf{posInt} \Leftarrow \mathsf{int} \rangle^{p_1}$. The cast for the domain of the function will attempt to cast the domain of the target to the domain of the source as follows: $\langle \mathsf{int} \Leftarrow \mathsf{posInt} \rangle^{\overline{p_1}}$. Preserving order for the range and reversing order for the domain is similar to the standard approach to function subtyping which is covariant in the range and contravariant in the domain.

The range cast retains the blame label p_1 because if this cast fails it is the fault of the function f. For example, a function f that maps all integer inputs to the integer -1 will cause the cast to fail and trigger the blame label. The blame label for the domain cast is the complement of the original blame label p_1 because if this cast fails, then it is the fault of the context for supplying an invalid argument x to the function f. However, we note that the type checker will guarantee that x has type posInt and the cast from posInt to int will always suceed. This means that the blame label $\overline{p_1}$ will never be blamed. The only situation where blame can occur is if the range cast fails to cast the less precise type int to the more precise type posInt with blame label p_1.

Consider the following cast which takes a function with a domain and range of type posInt to a less precise domain and range.

$$(\langle \mathsf{int} \to \mathsf{int} \Leftarrow \mathsf{posInt} \to \mathsf{posInt} \rangle^{p_2} f) x$$

As shown above, this cast will decompose into a cast for the range, $\langle \mathsf{int} \Leftarrow \mathsf{posInt} \rangle^{p_2}$, and domain, $\langle \mathsf{posInt} \Leftarrow \mathsf{int} \rangle^{\overline{p_2}}$, of the function. The range cast will always succeed, so the blame label p_2 will never be blamed. The only situation where blame can occur is if the domain cast fails to cast the less precise type int to the more precise type posInt with blame label $\overline{p_2}$. In this situation, the blame lies with the context containing the cast, as opposed to the cast itself.

In both of these instances, Wadler and Findler [38] prove that blame always lies with the less-precisely typed-code. When validating contracts expressed as type refinements, described in Section 5, we prove a similar theorem. More comprehensive theorems about the correctness of blame assignment have been proposed by Dimoulas et al. [10, 11]. Subsequent work on gradual typing that considers systems with both static and dynamic typing also uses "blame always lies with the less-precisely typed code" as a criteria for correctness. For instance, Ahmed et al. [1] developed a blame calculus for a language that integrates parametric polymorphism with static and dynamic typing. Fennell and Thiemann [13] proved a blame theorem for a linear lambda calculus with type Dynamic. Most recently, Wadler [37] surveys the history of the blame calculus and presents the latest developments. Keil and Thiemann [24] develop a blame assignment for higher order contracts that includes intersection and union contracts. Siek et al. [31] develop

three calculi for gradual typing and relate them in an effort to unite the concepts of blame and coercion.

Chapter 3

Session Types as Contracts

In a concurrent setting, there are two important reasons to consider dynamic monitoring of communication. The first is that when spawning a new process, part of the execution of a program now escapes immediate control of the original process. If the new process is compromised by a malicious intruder, then incorrect yet unchecked messages can wreak havoc on the original process. A second reason is that session types are explicitly designed to abstract away from local computation. This means we can use session types to safely connect communicating processes written in a variety of different languages, as long as they (dynamically!) adhere to the session protocol and basic data formats. However, designing monitoring infrastructure that allows precise blame assignment in the presence of higher-order processes is challenging. In such settings, channels of arbitrary type can be passed along other channels. When this occurs, the runtime monitor cannot immediately determine if the process communicating over the channel being passed along satisfies session fidelity, but must monitor further communication over this channel.

In this chapter, we present a model that dynamically monitors communication to enforce adherence to session types in a higher-order setting. We place a monitor on each channel that checks whether messages are consistent with the communication contract on that channel. If the message is determined to violate the contract, the monitor raises an alarm. When an alarm is raised, we are able to assign blame and prove one of an indicated set of possible culprits must have been compromised. We also prove that dynamic monitoring does not change system behavior for well-typed processes.

3.1 Model

We define an "attack" scenario when a process has been compromised and deviates from its prescribed session type. We use runtime monitors to detect such deviations and attribute blame to rogue processes. In this section, we discuss the adversary and trust model and explain the monitor design. We then formally define the operational semantics for the monitoring mechanism and for the blame assignment.

Trust Model We assume that processes are distributed across the network and communicate with each other by message-passing. We assume that there is a secure (trusted) network layer which ensures that messages are sent and received without error. In contrast, *all processes are untrusted*; any process could be compromised by an attacker.

Monitor capabilities We assume that the monitor can inspect communications between processes to check session fidelity, but it cannot observe internal operations of the executing processes. Only send, receive, spawn (cut), and forward (identity) requests can be seen by the monitor. This design decision is important because it allows our monitoring techniques to be applied in the situation where we make no assumptions about the internal structure of the communicating processes. The monitor is also trusted.

Monitors can raise alarms and assign blame when messages sent over channels are of the *wrong type*, which we explain in detail below. If a protocol violation is detected and alarm is raised, the computation is aborted.

Adversary capabilities We assume that channels are *private* in that only the processes at the two endpoints of a channel can send to or receive from it. Further, channel names are capabilities that are hard to forge. An attacker only knows the channel names that are given to it by the trusted runtime (e.g., through spawning a new process).

We define the following transition rule (named havoc) to represent an attacker's action of taking control of a process. The attacker replaces the original process with one of the attacker's choice. However, the attacker cannot forge channel names, and therefore, the set of free channel names in Q is a subset of that in P.

$$\mathsf{havoc} : \mathsf{proc}(c, P) \otimes !(\mathsf{fn}(P) \supseteq \mathsf{fn}(Q)) \longrightarrow \{\mathsf{proc}(c, Q)\}$$

Finally, because processes are untrusted, they cannot raise an alarm.

Monitor Semantics We have two variations of the monitoring semantics that allow us to provide different levels of precision in our blame assignment based on the assumptions we make. In the *Unverified-Spawn* semantics, we assume that we do not have access to the source code of a process being spawned before it starts executing. This assumption is conservative and treats all processes as black boxes. With these assumptions, our blame assignment returns a set of processes, where one of the indicated processes must have made a havoc transition and triggered the alarm. These semantics are described in detail in Jia et al. [23].

In the *Verified-Spawn* semantics, we assume that a spawned process can be statically typechecked against a given type to ensure that it is compatible with the channel it is being spawned on. If the spawned processes is not well-typed, all computation stops. If this check succeeds, then we are able to immediately absolve the spawning process. This allows us to provide more precise blame assignment by blaming a single process when an alarm is raised. In this chapter, we will present the simpler *Verified-Spawn* semantics.

We augment the operational semantics presented in Section 2.1 to include monitor actions. A subset of the rules are shown in Figure 3.2. We use the predicate $\mathsf{typecheck}(P :: x : A)$ to perform a static check that verifies that the process P is providing a service compatible with the

channel x of type A. We also augment our semantics to track which channels every process uses, using the notation $\mathsf{proc}(c, CH, P)$ to mean that process P provides a service along channel c and uses the channels in set CH. We carry these sets of channels throughout the computation. The other monitoring actions, denoted by gray boxes in Figure 3.2, either assert that a certain channel is in the set of channels that a process uses, or assert that a channel has a certain type. We use the notation $\neg(!(p), !(q))$ to mean that it is not the case that p and q are both true.

In the id rule, the monitor checks that channels a and b have the same type and that the channel b is used by the forwarding process. In the $\mathsf{id_a}$ rule, if any of the above conditions are not met, the system will raise an alarm. In the lolli_s rule, the monitor ensures that both the channels a and c_i are used by the sending process. The monitor also renames channel $c_i : A_1 \multimap A_2$ to channel $c_{i+1} : A_2$ once the sending step of the computation is complete. Similarly, in the lolli_r rule, the monitor renames channel $c_i : A_1 \multimap A_2$ to channel $c_{i+1} : A_2$. It also adds channel a to the set of channels that the resulting process, which provides a service on channel c_{i+1}, uses. In the cut rule, the monitor validates the spawned process and ensures that the set of channel names in P is a subset of those used by the spawning process with the exception of the fresh channel a_0. If the conditions are met, a new process P is spawned on channel a_0. The spawning process continues to execute and uses the channels not used by P.

Blame assignment When an alarm ($\mathsf{alarm}(a)$) is raised, the monitor assigns blame to exactly one process that provides a service along channel a. Informally, exactly that one process must have "havoced"; otherwise, type preservation will ensure that no alarm is raised.

Though we rename channels at every step of the computation, we are able to blame a single channel. This is the case because each time a channel a_i is renamed, its index is incremented and it is now called a_{i+1}. Therefore, when channel a_i is blamed, we can collapse all of the a_i's by just erasing the index and just blame channel a.

```
CameraFun : {Cam}
c ← CameraFun =
  case c of
  | take ⇒ pm ← recv c ;
    case pm of
    | once ⇒ wait pm ; picH ← takePic ; send c picH ; c ← CameraFun

ToSnap : {Snap ← User ; Cam}
s ← ToSnap ← u, c =
  c.take ; u.picPerm ;
  case u of
  | fail ⇒ c.fail ; s ← ToSnap ← u, c
  | succ ⇒ c.succ ; s.share ; perm ← recv u ; send c perm ;
    picH ← recv c  ; send s picH ; s ← ToSnap ← u, c
```

Figure 3.1: Snapchat

3.2 Examples

In this example, we illustrate monitoring with a mobile photosharing application, `Snapchat`, that takes and shares a user's photos and sends them to some remote entity. To take photos, `Snapchat` needs to operate the camera. To prevent the `Snapchat` application from continuously taking and sharing the user's photos, the camera requires that the user grant `Snapchat` permission every time `Snapchat` wants to take and share a photo. This example contains three main processes: the `Snapchat` application process, the camera process, and the user process.

Types and Encoding We encode the expected behavior of each process as a session type declaration below.

$$\begin{aligned}
&\mathsf{stype}\ \ \mathsf{Cam} = \&\{\mathsf{take} : \mathsf{photoPerm} \multimap \&\{\mathsf{fail} : \mathsf{Cam}\ ; \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{succ} : \mathsf{picHandle} \otimes \mathsf{Cam}\}\} \\
&\mathsf{stype}\ \ \mathsf{User} = \&\{\mathsf{picPerm} : \oplus\{\mathsf{fail} : \mathsf{User}; \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{succ} : \mathsf{photoPerm} \otimes \mathsf{User}\}\} \\
&\mathsf{stype}\ \ \mathsf{photoPerm} = \oplus\{\mathsf{once} : \mathbf{1}\} \\
&\mathsf{stype}\ \ \mathsf{Snap} \qquad = \oplus\{\mathsf{share} : \mathsf{picHandle} \otimes \mathsf{Snap}\}
\end{aligned}$$

Camera After the client selects take, the camera process requires the client to send a photo permission channel of type photoPerm before sending a handle to a picture to the client. If the user has not granted the permission, the request fails and the camera process continues to offer a service of type Cam. Otherwise, after the request succeeds and the picture handle is sent, the camera process continues to offer a service of type Cam.

User When a process needs permission to access the camera, it communicates with the user process and selects picPerm. If the user sends the fail label, the user process continues to offer a service of type User, without granting its client permission to use the camera. If the user process sends a succ label, it then spawns a new process that provides a service of type photoPerm and sends the new process' channel to its client. The type photoPerm is an internal choice, labeled once.

Snapchat The ToSnap process uses c to communicate with the camera process and u to communicate with the user process and offers the picture sharing service along channel s. The process first instructs the camera take a picture and then asks the user process for permission. If the user does not grant the permission then no picture is sent and the `ToSnap` process continues to try and send a picture. If the user grants the permission, the ToSnap process sends its client the label share. It then receives a channel connecting to a permission process from the user, and forwards the channel to the camera. Finally, the ToSnap process receives a picture handle from the camera, and sends it to `ToSnap`'s client.

The CameraFun and ToSnap processes are shown in Figure 3.1. We assume that the takePic function returns a picture handle.

Monitoring Scenarios

We show two monitoring scenarios to demonstrate how our monitor can detect violations of invariants specified by the session types. In these scenarios, an attacker tries to take pictures without being granted permissions required by the camera.

Scenario 1 The `ToSnap` process is compromised by an attacker. The havoced process does not ask for permission from the user and instead of sending a permission to the camera, sends an integer value (i.e. replacing lines $perm \leftarrow \mathsf{recv}\ u\ ;\ \mathsf{send}\ c\ perm$ of the ***ToSnap*** process with $\mathsf{send}\ c\ n$).

The monitor on channel c is expecting a value of type PhotoPerm which should be a channel. The monitor will try to typecheck the integer n as a channel and fail. It will then raise an alarm ($\mathsf{alarm}(s)$). Here, blame is assigned to one process, ToSnap (offering along channel s).

Scenario 2 The ToSnap process is working appropriately and has gotten a legitimate photo permission from the user. When it sends the permission to the camera, the sent process is taken over by an attacker. Instead of $\mathsf{proc}(d, d \leftarrow pm)$, where the process offering along channel pm is spawned by the user, the attacker changes it to $\mathsf{proc}(d, \mathsf{send}\ d\ n)$. For simplicity, we assume that pm is a special channel designated for photo permissions. The monitor will raise an alarm ($\mathsf{alarm}(d)$) when the above compromised process tries to send an integer value to d, because the monitor is expecting the label once. Because s spawned d, blame is assigned to both s and d. One interesting point is that in this scenario, `Snapchat` actually has the right permission. We note that this exploit is only possible in the *Unverified-Spawn* system and not the *Verified-Spawn* system.

3.3 Metatheory

We identify three high-level properties that the monitor should satisfy: correctness of the blame assignment, the fact that well-behaved processes are not blamed, and transparency of the monitor.

The correctness of the blame assignment is defined as follows. Let the context Ω be the multiset of processes and messages describing the current state of computation. We define the context Γ to map channels in the system to their types. We say that it is correct to blame a set of processes if at least one of the processes in the set has made a havoc transition. We write $\models \Omega : \mathsf{wf}$ to denote that the state Ω is well-typed. This well-typedness requires that all processes and messages in Ω be typed using typing rules in Figure 5.7.

Definition 1 (Correctness of blame)**.** *A channel a is correct to be blamed w.r.t. the execution trace $\mathcal{T} = \Gamma, \Omega \longrightarrow^* \mathsf{alarm}(a_i)$ with $\models \Omega : \mathsf{wf}$ if the process providing a service on a_i has made a havoc transition in $\mathcal{T}$.*

Second, if all processes are well-typed to begin with and no process is compromised at run-time, then the monitor should not raise an alarm. This property shows that a havoc transition is necessary for the monitor to halt the execution and assign blame.

Definition 2 (Well-typed configurations do not raise alarms)**.** *Given any* $\mathcal{T} = \Gamma, \Omega \longrightarrow^* \Gamma', \Omega'$ *such that* $\models \Omega : \mathsf{wf}$ *and* $\mathcal{T}$ *does not contain any havoc transitions, there does not exist an* a *such that* $\mathsf{alarm}(a) \in \Omega'$.

Finally, the monitor should not change the behavior of well-typed processes. We write $\longrightarrow^-$ to denote the operational semantics without the monitor actions. If the initial configuration is well-typed and no process is compromised, then executing the configuration with and without the monitor should yield the same result.

Definition 3 (Monitor transparency)**.** *Given any* $\mathcal{T} = \Gamma, \Omega \longrightarrow^* \Gamma', \Omega'$ *such that* $\models \Omega : \mathsf{wf}$ *and* $\mathcal{T}$ *does not contain any havoc transitions. Then* $\Omega(\longrightarrow^-)^*\Omega'$.

Proposed Work Using an older version of the semantics, which used message queues, we have proved that the properties defined above hold for both the *Unverified-Spawn* and *Verified-Spawn* systems. The proofs for the *Unverified-Spawn* system are available in Jia et al [22].

We have also proved that the above properties hold in a shared setting where processes can be persistent (written $!\mathsf{proc}(c, P)$) and create linear copies of themselves. The addition of sharing also requires dynamic monitoring because process copying causes part of the execution of a program to escape immediate control of the original process.

As part of the proposed work, we will prove these properties using the semantics presented in Section 2.1 for both the *Unverified-Spawn* and *Verified-Spawn* systems.

3.4 Related Work

Compared to the body of work mentioned in Section 2.2, our work focuses on systems where processes communicate with each other via message-passing. At a high-level, we can relate our adversary model to the work on blame assignment as follows. Each process can be viewed as a program written in dynamically typed language. Our monitor enforces the coercion of session types by observing the communications between the processes. Our blame assignment always includes the compromised process. If we view the compromised process as a less-precisely-typed program, our correctness of blame property is similar to the notion proposed in Wadler and Findler [38]: blame always falls on less-precisely-typed programs.

The work most closely related to ours is on multi-party session types. Bocchi et al. [3] and Chen et al. [7] assume a similar asynchronous message passing model as ours. Their monitor architecture is also similar to ours; monitors are placed at the ends of the communication channels and monitor communication patterns. One key difference is that their monitors do not raise alarms; instead, the monitors suppress bad messages and move on. Our monitors halt the execution and assign blame. Consequently, this work does not have theorems about blame assignment which are central to our work. Using global types, their monitors can additionally enforce global properties such as deadlock freeness, which our monitors cannot. Our work supports higher-order processes, that is, processes that can spawn other processes and delegated communication to other processes, while their work does not.

The recently-developed Whip system [39] addresses a similar problem our work, but does not use session types. They use a dependent type system to implement a contract monitoring system

that can connect services written in different languages. Their system is also higher order, and allows processes that are monitored by Whip to interact with unmonitored processes.

$(c_i)^+ = c_{i+1}$

one_s : $\mathsf{proc}(a, CH, \mathsf{close}\ a) \longrightarrow \mathsf{msg}(a, \mathsf{close}\ a)$

one_r : $\mathsf{msg}(a, \mathsf{close}\ a), \mathsf{proc}(c, CH, \mathsf{wait}\ a'; Q), !(a \in CH) \longrightarrow \mathsf{proc}(c, CH, Q)$

with_s : $\mathsf{proc}(d, CH, c_i.k; Q) \longrightarrow \mathsf{proc}(d, CH, [c_i^+/c_i]Q), \mathsf{msg}(c_i^+, c_i.k; c_i^+ \leftarrow c_i)$

with_r : $\mathsf{msg}(c_i^+, c_i.k; c_i^+ \leftarrow c_i), \mathsf{proc}(c_i, CH, \mathsf{case}\ c_i\{\ell \Rightarrow P_\ell\}_{\ell \in L}) \longrightarrow \mathsf{proc}(c_i^+, CH, [c_i^+/c_i]P_k)\}$

plus_s : $\mathsf{proc}(c_i, CH, c_i.k; P) \longrightarrow \mathsf{msg}(c_i, c_i.k; c_i \leftarrow c_i^+), \mathsf{proc}(c_i^+, CH, [c_i^+/c_i]P)$

plus_r : $\mathsf{msg}(c_i, c_i.k; c_i \leftarrow c_i^+), \mathsf{proc}(d, CH, \mathsf{case}\ c_i\{\ell \Rightarrow Q_\ell\}_{\ell \in L}) \longrightarrow \mathsf{proc}(d, CH, [c_i^+/c_i]Q_k)$

id : $\mathsf{proc}(a, CH, a \leftarrow b; \mathcal{C}), !(b \in CH), !(a : A), !(b : A) \longrightarrow [b/a]\mathcal{C}$

id_a : $\mathsf{proc}(a, CH, a \leftarrow b; \mathcal{C}), \neg(!(b \in CH), !(a : A), !(b : A)) \longrightarrow \mathsf{alarm}(a)$

lolli_s : $\mathsf{proc}(d, CH, \mathsf{send}\ c_i\ a; Q), !(c_i \in CH), !(a \in CH), !(c_i : A_1 \multimap A_2))$
$\longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i), \mathsf{proc}(d, CH \setminus a, [c_i^+/c_i]Q), !(c_i^+ : A_2)$

lolli_s_a : $\mathsf{proc}(d, CH, \mathsf{send}\ c_i\ a; Q), \neg(!(c_i \in CH), !(a \in CH), !(c_i : A_1 \multimap A_2))$
$\longrightarrow \mathsf{alarm}(d)$

lolli_r : $\mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i), \mathsf{proc}(c_i, CH, x \leftarrow \mathsf{recv}\ c_i; P), !(c_i : A_1 \multimap A_2)$
$\longrightarrow \mathsf{proc}(c_i^+, CH \cup a, [c_i^+/c_i][a/x]P), !(c_i^+ : A_2)$

lolli_r_a : $\mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i), \mathsf{proc}(c_i, CH, x \leftarrow \mathsf{recv}\ c_i; P), !(c_i :\not A_1 \multimap A_2)$
$\longrightarrow \mathsf{alarm}(c_i)$

tensor_s : $\mathsf{proc}(c_i, CH, \mathsf{send}\ c_i\ a; P), !(c_i : A_1 \otimes A_2), !(a \in CH)$
$\longrightarrow \mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+), \mathsf{proc}(c_i^+, CH \setminus a, [c_i^+/c_i]P), !(c_i : A_2)$

tensor_s_a : $\mathsf{proc}(c_i, CH, \mathsf{send}\ c_i\ a; P), \neg(!(c_i : A_1 \otimes A_2), !(a \in CH))$
$\longrightarrow \mathsf{alarm}(c_i)$

tensor_r : $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+, \mathsf{proc}(d, CH, x \leftarrow \mathsf{recv}\ c_i; Q), !(c_i \in CH) \otimes !(c_i : A_1 \otimes A_2)$
$\longrightarrow \mathsf{proc}(d, CH \cup a, [c_i^+/c_i][a/x]Q), !(c_i : A_2)$

tensor_r_a : $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+, \mathsf{proc}(d, CH, x \leftarrow \mathsf{recv}\ c_i; Q), \neg(!(c_i \in CH) \otimes !(c_i : A_1 \otimes A_2))$
$\longrightarrow \mathsf{alarm}(d)$

cut : $\mathsf{proc}(c, CH, x : A \leftarrow P; Q), !(\mathsf{typecheck}(P :: x : A)), !(fcc(P) \subset (CH \setminus a_0))$
$\longrightarrow \mathsf{proc}(c, CH/\mathsf{fn}(b), [a_0/x]Q), \mathsf{proc}(a_0, \mathsf{fn}(P), [a_0/x]P), !(a_0 : A) \quad (a_0\ fresh)$

cut_a : $\mathsf{proc}(c, CH, x : A \leftarrow P; Q), \neg(!(\mathsf{typecheck}(P :: x : A)), !(\mathsf{fn}(P) \subset (CH \setminus a)))$
$\longrightarrow \mathsf{alarm}(c)$

Figure 3.2: Verified-Spawn Monitor Rules.

Chapter 4

Partial Identity Processes as Behavioral Contracts

The previous chapter of this proposal presented contracts, specified as session-types, that enforced communication protocols between processes. In that setting, we assigned each channel a monitor for detecting whether messages observed along the channel adhere to the prescribed session type. The monitor can then detect any deviant behavior the processes exhibit and trigger alarms. However, contracts based solely on session types are inherently limited in their expressive power. Many contracts that we would like to enforce cannot even be stated using session types alone. As a simple example, consider a "factorization service" which may be sent a (possibly large) integer x and is supposed to respond with a list of prime factors. Session types can only express that the request is an integer and the response is a list of integers, which is insufficient.

In this chapter, we show that by generalizing the class of monitors beyond those derived from session types, we can enforce, for example, that multiplying the numbers in the response yields the original integer x. To handle these contracts, we have designed a model where our monitors execute as transparent processes alongside the computation. They are able to maintain internal state which allows us to check complex properties. These monitoring processes act as partial identities, which do not affect the computation except possibly raising an alarm, and merely observe the messages flowing through the system. They then perform whatever computation is needed, for example, they can compute the product of the factors, to determine whether the messages are consistent with the contract. If the message violates the contract, they stop the computation. In this chapter, we present a method for verifying that monitors are truly partial identities and examples illustrating the breadth of contracts that our monitors can enforce.

4.1 Model

As a first simple example, let's take a process that receives one positive integer n and factors it into two integers p and q that are sent back where $p \leq q$. The part of the specification that is *not* enforced is that if n is not prime, p and q should be proper factors, but we at least enforce that all

numbers are positive and $n = p * q$.

$$
\begin{array}{l}
\mathsf{factor\_t} = \forall n{:}\mathsf{int}.\, \exists p{:}\mathsf{int}.\, \exists q{:}\mathsf{int}.\, \mathbf{1} \\
\mathsf{factor\_monitor} : \{\mathsf{factor\_t} \leftarrow \mathsf{factor\_t}\} \\
c \leftarrow \mathsf{factor\_monitor} \leftarrow d = \\
\quad n \leftarrow \mathsf{recv}\ c\ ;\ \mathsf{assert}\ l_1\ (n > 0)\ ;\ \mathsf{send}\ d\ n\ ; \\
\quad p \leftarrow \mathsf{recv}\ d\ ;\ \mathsf{assert}\ l_2\ (p > 0)\ ;\ q \leftarrow \mathsf{recv}\ d\ ;\ \mathsf{assert}\ l_3\ (q > 0)\ ;\ \mathsf{assert}\ l_4\ (p \leq q)\ ; \\
\quad \mathsf{assert}\ l_5\ (n = p * q)\ ;\ \mathsf{send}\ c\ p\ ;\ \mathsf{send}\ c\ q\ ;\ c \leftarrow d
\end{array}
$$

To check that factor_monitor is a partial identity we need to track that p and q are received from the provider, in this order. In general, for any received message, we need to enter it into a message queue q and we need to check that the messages are passed on in the correct order. As a first cut (to be generalized several times), we write for negative types:

$$[q](b : B^-)\ ;\ \Psi \vdash P :: (a : A^-)$$

which expresses that the two endpoints of the monitor are $a : A^-$ and $b : B^-$ (both negative), and we have already received the messages in queue q along channel a.

A monitor, at the top level, is defined with

$$
\begin{array}{l}
\mathit{mon} : \tau_1 \to \ldots \to \tau_n \to \{A \leftarrow A\} \\
a \leftarrow \mathit{mon}\ x_1 \ldots x_n \leftarrow b = P
\end{array}
$$

where the context Ψ declares value variables x. The body P here is type-checked as one of (depending on the polarity of A)

$$[\,](b : A^-)\ ;\ \Psi \vdash P :: (a : A^-) \quad \text{or} \quad (b : A^+)\ ;\ \Psi \vdash P :: [\,](a : A^+)$$

where $\Psi = (x_1{:}\tau_1) \cdots (x_n{:}\tau_n)$. A use such as

$$c \leftarrow \mathit{mon}\ e_1 \ldots e_n \leftarrow c$$

is transformed into

$$c' \leftarrow \mathit{mon}\ e_1 \ldots e_n \leftarrow c\ ;\ c \leftarrow c'$$

for a fresh c' and type-checked accordingly.

In general, queues have the form $q = m_1 \cdots m_n$ with

m	::=	l_k	labels	$\oplus, \&$				
	$\mid$	c	channels	$\otimes, \multimap$	$\mid$	n	value variables	$\exists, \forall$
	$\mid$	end	close	$\mathbf{1}$	$\mid$	shift	shifts	$\uparrow, \downarrow$

where m_1 is the front of the queue and m_n the back.

When P receives a message, we add it to the end of the queue q. We also need to add it to the context Ψ to remember its type. In our example $\tau = \mathsf{int}$.

$$\frac{[q \cdot n](b : B)\ ;\ \Psi, n{:}\tau \vdash P :: (a : A^-)}{[q](b : B)\ ;\ \Psi \vdash n \leftarrow \mathsf{recv}\ a\ ;\ P :: (a : \forall n{:}\tau.\, A^-)}\ \forall R$$

Conversely, when we *send* along channel b the message must be equal to the one at the front of the queue (and therefore it must be a variable). The value variable $m$remains in the context so it can be reused for later assertion checks. However, it can never be sent again since it has been removed from the queue.

$$\frac{[q](b : [m/n]B)\ ;\ \Psi, m{:}\tau \vdash Q :: (a : A)}{[m \cdot q](b : \forall n{:}\tau.\ B)\ ;\ \Psi, m{:}\tau \vdash \mathsf{send}\ b\ m\ ;\ Q :: (a : A)}\ \forall L$$

The rest of the rules characterizing partial identity processes are presented in Gommerstadt et al [16]. All the examples described in the next section can be verified to be partial identities under our definition.

4.2 Examples

In this section, we present a variety of monitoring processes that can enforce various contracts. Our examples will mainly be concerned with lists using the definition from Section 2.1. Any monitor that enforces a contract on a list must peel off each layer of the type one step at a time (by sending or receiving over the channel as dictated by the type), perform the required checks on values or labels, and then reconstruct the original type (again, by sending or receiving as appropriate). Since a minimal number of shifts can be inferred during elaboration of the syntax [29], we suppress them in the examples.

Refinement The simplest kind of monitoring process we can write is one that models a refinement of an integer type (shown in Figure 4.1); for example, a process that checks whether every element in the list is positive. This is a recursive process that receives the head of the list from channel b, checks whether it is positive (if yes, it continues to the next value, if not it aborts), and then sends the value along to reconstruct the monitored list over channel a.

$\mathsf{pos\_mon} : \{\mathsf{list} \leftarrow \mathsf{list}\}$
$a \leftarrow \mathsf{pos\_mon} \leftarrow b =$
  $\mathsf{case}\ b\ \mathsf{of}$
  $\mid\ \mathsf{nil} \Rightarrow a.\mathsf{nil}\ ;\ \mathsf{wait}\ b\ ;\ \mathsf{close}\ a$
  $\mid\ \mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ b\ ;\ \mathsf{assert}\ l\ (x > 0)\ ;\ a.\mathsf{cons}\ ;\ \mathsf{send}\ a\ x\ ;\ a \leftarrow \mathsf{pos\_mon} \leftarrow b$

Figure 4.1: Integer refinement

$\mathsf{empty\_t} = \oplus\{\mathsf{nil} : \mathbf{1}\}$
$\mathsf{empty\_mon} : \{\mathsf{empty\_t} \leftarrow \mathsf{list}\}$
$a \leftarrow \mathsf{empty\_mon} \leftarrow b =$
  $\mathsf{case}\ b\ \mathsf{of}$
  $\mid\ \mathsf{nil} \Rightarrow \mathsf{wait}\ b\ ;\ a.\mathsf{nil}\ ;\ \mathsf{close}\ a$
  $\mid\ \mathsf{cons} \Rightarrow \mathsf{abort}\ l$

$\mathsf{non\_empty\_t} = \oplus\{\mathsf{cons} : \exists n{:}\mathsf{int}.\mathsf{list}\}$
$\mathsf{nonempty\_mon} : \{\mathsf{non\_empty\_t} \leftarrow \mathsf{list}\}$
$a \leftarrow \mathsf{nonempty\_mon} \leftarrow b =$
  $\mathsf{case}\ b\ \mathsf{of}$
  $\mid\ \mathsf{nil} \Rightarrow \mathsf{abort}\ l$
  $\mid\ \mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ b\ ;\ a.\mathsf{cons}\ ;\ \mathsf{send}\ a\ x\ ;\ a \leftarrow b$

Figure 4.2: Label refinement

Our monitors can also exploit information contained in the labels present in the external and internal choices. We show two examples of monitors that model label refinement in Figure 4.2. The empty_mon process checks whether the list offered over channel b is empty and aborts if channel b sends the label cons. Similarly, the nonempty_mon monitor checks whether the list offered over channel b is not empty and aborts if channel b sends the label nil. These two monitors enforce refinements: $\{\mathsf{nil}\} \subseteq \{\mathsf{nil}, \mathsf{cons}\}$ and $\{\mathsf{cons}\} \subseteq \{\mathsf{nil}, \mathsf{cons}\}$.

Monitors with internal state We now move beyond refinement contracts, and model contracts that have to maintain some internal state. We present a monitor that checks whether a set of right and left parentheses match (shown in Figure 4.3). The match_mon monitor uses its internal state to push every left parenthesis it sees on its stack and to pop it off when it sees a right parenthesis. For brevity, we model our list of parentheses by marking every left parenthesis with a 1 and right parenthesis with a -1. So the sequence ()()) would look like $1, -1, 1, -1, -1$. As we can see, this is not a proper sequence of parentheses because adding all of the integer representations does not yield 0. In a similar vein, we can implement a process that checks that a tree is serialized correctly, which is related to recent work on context-free session types by Thiemann and Vasconcelos [33].

```
match_mon : int → {list ← list}
a ← match_mon count ← b =
  case b of
  | nil ⇒ assert l1(count = 0) ; a.nil ; wait b ; close a
  | cons ⇒ x ← recv b ; a.cons ;
    if (x = 1) then send a x ; a ← match_mon (count + 1) ← b;
    else if (x = −1) then assert l2 (count > 0) ; send a x ; a ← match_mon (count − 1) ← b ;
    else abort l3   //invalid input
```

Figure 4.3: Parenthesis Matching monitor

We summarize the various invariants our monitors can check, and state maintained by the monitors, in Table 4.1.

Table 4.1: Example Summary

Contract	**Monitor State**
List of integers is in ascending order	Largest integer seen so far
List of integers corresponds to correctly serialized binary tree	Stack - push for node, pop for leaf
Sorting procedure permutes elements of a list	Sum of hash values of each element
Doubling process mapped over a list of integers is monotonic	Input integer
Factoring procedure outputs integers that multiply to input	Input Integer

4.3 Metatheory

We prove that the partial-identity criterion presented in Section 4.1 guarantees that session-typed monitoring processes are observationally equivalent to partial identity processes. A simplified version of the theorem is stated below (a more complete version and proof details can be found in Gommerstadt et al [16]).

Theorem 1 (Transparency)**.** *Let P be a process that adheres to the typing rules presented in Gommerstadt et al [16]. Then,* $b : B \vdash \mathsf{proc}(b, a \leftarrow b) \sim \mathsf{proc}(a, P) :: (a : A)$.

Proof. We prove the theorem by first defining a notion of observational equivalence for messages. We then contruct a weak bisimulation. We note that this bisimulation is not standard because the process P can terminate due to a failed assertion. We only require observational equivalence up to termination. □

4.4 Related Work

Most closely related to our is the work by Disney et al. [12], which investigates behavioral contracts that enforce temporal properties for modules. Our contracts (i.e., session types) enforce temporal properties as well; the session types specify the order in which messages are sent and received by the processes. Our contracts can also make use of internal state, as those of Disney et al. do, but our system is concurrent, while their system does not consider concurrency.

Recently, Melgratti and Padovani have developed chaperone contracts for higher-order session types [25]. Their work is based on a classic interpretation of session types, instead of an intuitionistic one like ours; therefore, they do not handle spawning or forwarding processes. While their contracts also inspect messages passed between processes, unlike ours, they cannot model contracts which rely on the monitor making use of internal state (e.g., the parenthesis matching). They proved a blame theorem relying on the notion of locally correct modules, which is a semantic categorization of whether a module satisfies the contract. We did not prove a general blame theorem; instead, we prove a somewhat standard safety theorem for cast-based contracts (discussed in Chapter 5).

The Whip system [39] uses a dependent type system to implement a contract monitoring system. However, Whip cannot handle stateful contracts. Another distinguishing feature of our monitors is that they are partial identity processes encoded in the same language as the processes to be monitored.

Chapter 5

Refinement Types as Contracts

In this chapter, we show how to check refinement types dynamically using our partial-identity monitors. We encode refinements as type casts, which allows processes to remain well-typed with respect to the non-refinement type system (described in Section 2.1). These casts are translated at run time to monitors that validate whether the cast expresses an appropriate refinement. If so, the monitors behave as identity processes; otherwise, they raise an alarm and abort. For refinement contracts, we can prove a safety theorem, analogous to the classic "Well-typed Programs Can't be Blamed" [38], stating that if a monitor enforces a contract that casts from type A to type B, where A is a subtype of B, then this monitor will never raise an alarm.

5.1 Surface Language

We first augment messages and processes to include casts as follows. We write $\langle A \Leftarrow B \rangle^\rho$ to denote a cast from type B to type A, where ρ is a unique label for the cast. The cast for values is written as ($\langle \tau \Leftarrow \tau' \rangle^\rho$). In this section, we treat the types τ' and τ as refinement types of the form $\{n{:}t \mid b\}$, where b is a boolean expression that expresses simple properties of the value n. We express any unrefined type τ by writing $\{n : t \mid \mathsf{true}\}$ which is compatible with earlier sections of this proposal.

$$P \quad ::= \quad \cdots \mid x \leftarrow \langle \tau \Leftarrow \tau' \rangle^\rho \; v \; ; \; Q \mid a{:}A \leftarrow \langle A \Leftarrow B \rangle^\rho \; b$$

Both of the additional rules to type casts are shown below. We only allow casts between two types that are compatible with each other (written $A \sim B$), which is co-inductively defined based on the structure of the types. The full definition is shown in Gommerstadt et al [16].

$$\frac{A \sim B}{\Psi \; ; \; b : B \vdash a \leftarrow \langle A \Leftarrow B \rangle^\rho \; b :: (a : A)} \; \mathsf{id\_cast}$$

$$\frac{\Psi \vdash v : \tau' \quad \Psi, x : \tau \; ; \; \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'}{\Psi \; ; \; \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau' \rangle^\rho \; v \; ; \; Q :: (c : C)} \; \mathsf{val\_cast}$$

5.2 Examples

We present two examples – one for an integer refinement and one for a label refinement. Consider the below cast:

$$\langle \{\exists n : \mathsf{int} \mid n > 2.A\}_a \Leftarrow \{\exists n : \mathsf{int} \mid n > 0.B\}_b \rangle^\rho$$

where channel a has type $\{\exists n : \mathsf{int} \mid n > 2.A\}$ and channel b has type $\{\exists n : \mathsf{int} \mid n > 0.B\}$. To validate this cast, we first receive the integer value from channel b. We then assert that this value meets channel a's refinement contract by checking if it is larger than two. We then send this value along to channel a and continue checking the rest of the type. The monitor would look like this:

$[[\langle \{\exists n : \mathsf{int} | n > 2.A\} \Leftarrow \{\exists n : \mathsf{int} | n > 0.B\} \rangle^\rho]]_{a,b} =$
  $x \leftarrow \mathsf{recv}\ b$;
  $\mathsf{assert}\ \rho\ (x > 2)$;
  $\mathsf{send}\ a\ x$;
  $[[\langle A \Leftarrow B \rangle^\rho]]_{a,b}$;

Figure 5.1: Integer cast translation

We can also handle casts with label refinements like the following:

$$\langle \oplus\{\mathsf{cons} : \exists n : \mathsf{int.list}; \mathsf{nil} : 1\}_a \Leftarrow \oplus\{\mathsf{cons} : \exists n : \mathsf{int.list}\} \rangle_b^\rho$$

In this example, we have to case on the possible labels received from channel b. There is only one option, a $cons$ label. In this case, we send the $cons$ label along to channel a.

$[[\langle \oplus\{\mathsf{cons} : \exists n : \mathsf{int.list}; \mathsf{nil} : 1\} \Leftarrow \oplus\{\mathsf{cons} : \exists n : \mathsf{int.list}\} \rangle^\rho]]_{a,b} =$
  $\mathsf{case}\ b\ \mathsf{of}$
  $|\ \mathsf{cons} \Rightarrow a.\mathsf{cons}$;
  $[[\langle \exists n : \mathsf{int.list} \Leftarrow \exists n : \mathsf{int.list} \rangle^\rho]]_{a,b}$

Figure 5.2: Label cast translation

5.3 Translation to Monitors

At runtime, casts are translated into monitoring processes. A cast $a \leftarrow \langle A \Leftarrow B \rangle^\rho\ b$ is implemented as a monitor. This monitor ensures that the process that offers a service on channel b behaves according to the prescribed type A. Because of the typing rules, we are assured that channel b must adhere to the type B.

Figure 5.3 is a summary of all the translation rules, except recursive types. The translation is of the form: $[\![\langle A \Leftarrow B \rangle^\rho]\!]_{a,b} = P$, where A, B are types; the channels a and b are the offering channel and monitoring channel (respectively) for the resulting monitoring process P; and ρ is

the label of the monitor (i.e., the contract). Here, the communication between processes is bi-directional. Though the blame is always triggered by processes sending messages to the monitor, our contracts may depend on a set of the values received so far, so it does not make sense to blame one party. Further, in the case of forwarding, the processes at either end of the channel are behaving according to the types (contracts) assigned to them, but the cast may connect two processes that have incompatible types. In this case, it is unfair to blame either one of the processes. Instead, we blame the label of the failed contract.

$$
\begin{array}{lcl}
\mathsf{one} & : & [\![\langle \mathbf{1} \Leftarrow \mathbf{1} \rangle^{\rho}]\!]_{a,b} = \mathsf{wait}\ b; \mathsf{close}\ a \\
\mathsf{lolli} & : & [\![\langle A_1 \multimap A_2 \Leftarrow B_1 \multimap B_2 \rangle^{\rho}]\!]_{a,b} = \\
 & & x \leftarrow \mathsf{recv}\ a\ ;\ y \leftarrow [\![\langle B_1 \Leftarrow A_1 \rangle^{\rho}]\!]_{y,x} \leftarrow x\ ;\ \mathsf{send}\ b\ y\ ;\ [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]_{a,b} \\
\mathsf{tensor} & : & [\![\langle A_1 \otimes A_2 \Leftarrow B_1 \otimes B_2 \rangle^{\rho}]\!]_{a,b} = \\
 & & x \leftarrow \mathsf{recv}\ b\ ;\ y \leftarrow [\![\langle A_1 \Leftarrow B_1 \rangle^{\rho}]\!]_{y,x} \leftarrow x\ ;\ \mathsf{send}\ a\ y\ ;\ [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]_{a,b} \\
\mathsf{forall} & : & [\![\langle \forall\{n : \tau \mid e\}.\, A \Leftarrow \forall\{n : \tau' \mid e'\}.\, B \rangle^{\rho}]\!]_{a,b} = \\
 & & n \leftarrow \mathsf{recv}\ a\ ;\ \mathsf{assert}\ \rho\, e'(x)\ \ ;\ \mathsf{send}\ b\ x; [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b} \\
\mathsf{exists} & : & [\![\langle \exists\{n : \tau \mid e\}.\, A \Leftarrow \exists\{n : \tau' \mid e'\}.\, B \rangle^{\rho}]\!]_{a,b} = \\
 & & n \leftarrow \mathsf{recv}\ b\ ;\ \mathsf{assert}\ \rho\, e(x)\ \ ;\ \mathsf{send}\ a\ x; [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b} \\
\mathsf{up} & : & [\![\langle \uparrow A \Leftarrow \uparrow B \rangle^{\rho}]\!]_{a,b} = \\
 & & \mathsf{shift} \leftarrow \mathsf{recv}\ b\ ;\ \mathsf{send}\ a\ \mathsf{shift}\ ;\ [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b} \\
\mathsf{down} & : & [\![\langle \downarrow A \Leftarrow \downarrow B \rangle^{\rho}]\!]_{a,b} = \\
 & & \mathsf{shift} \leftarrow \mathsf{recv}\ a\ ;\ \mathsf{send}\ b\ \mathsf{shift}\ ;\ [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b} \\
\mathsf{plus} & : & [\![\langle \oplus\{\ell : A_\ell\}_{\ell \in I} \Leftarrow \oplus\{\ell : B_\ell\}_{\ell \in J} \rangle^{\rho}]\!]_{a,b} = \\
 & & \mathsf{case}\ b\ (\ell \Rightarrow Q_\ell)_{\ell \in I} \\
 & & \mathsf{where}\ \forall \ell, \ell \in I \cap J,\ a.\ell\ ;\ [\![\langle A_\ell \Leftarrow B_\ell \rangle^{\rho}]\!]_{a,b} = Q_\ell \\
 & & \mathsf{and}\ \forall \ell, \ell \in J \wedge \ell \notin I,\ Q_\ell = \mathsf{abort}\ \rho \\
\mathsf{with} & : & [\![\langle \&\{\ell : A_\ell\}_{\ell \in I} \Leftarrow \&\{\ell : B_\ell\}_{\ell \in J} \rangle^{\rho}]\!]_{a,b} = \\
 & & \mathsf{case}\ a\ (\ell \Rightarrow Q_\ell)_{\ell \in I} \\
 & & \mathsf{where}\ \forall \ell, \ell \in I \cap J,\ b.\ell\ ;\ [\![\langle A_\ell \Leftarrow B_\ell \rangle^{\rho}]\!]_{a,b} = Q_\ell \\
 & & \mathsf{and}\ \forall \ell, \ell \in I \wedge \ell \notin J,\ Q_\ell = \mathsf{abort}\ \rho
\end{array}
$$

Figure 5.3: Cast translation

The translation is defined inductively over the structure of the types. The tensor rule generates a process that first receives a channel (x) from the channel being monitored (b), then spawns a new monitor to monitor x, making sure that it behaves as type A_1. Then, it passes the new monitor's offering channel y to channel a. Finally, the monitor continues to monitor channel b to make sure that it behaves as type A_2. The exists rule generates a process that first receives a value from the channel b, then checks the boolean condition e to validate the contract. The with rule generates a process that checks that all of the external choices promised by the type $\&\{\ell : A_\ell\}_{\ell \in I}$ are offered by the process being monitored. If a label in the set I is not implemented, then the monitor aborts with the label ρ.

$$\frac{}{1 \leq 1}\ 1 \qquad \frac{A \leq A' \quad B \leq B'}{A \otimes B \leq A' \otimes B'}\ \otimes \qquad \frac{A' \leq A \quad B \leq B'}{A \multimap B \leq A' \multimap B'}\ \multimap$$

$$\frac{A_k \leq A'_k \text{ for } k \in J \quad J \subseteq I}{\oplus\{\ell_k : A_k\}_{k \in J} \leq \oplus\{\ell_k : A'_k\}_{k \in I}}\ \oplus \qquad \frac{A_k \leq A'_k \text{ for } k \in J \quad I \subseteq J}{\&\{\ell_k : A_k\}_{k \in J} \leq \&\{\ell_k : A'_k\}_{k \in I}}\ \&$$

$$\frac{A \leq B}{\downarrow A \ \leq \downarrow B}\downarrow \qquad \frac{A \leq B}{\uparrow A \ \leq \uparrow B}\uparrow \qquad \frac{A \leq B \quad \tau_1 \leq \tau_2}{\exists n : \tau_1 . A \leq \exists n : \tau_2 . B}\ \exists \qquad \frac{A \leq B \quad \tau_2 \leq \tau_1}{\forall n : \tau_1 . A \leq \forall n : \tau_2 . B}\ \forall$$

$$\frac{\mathsf{def}(A) \leq \mathsf{def}(B)}{A \leq B}\ \mathsf{def} \qquad \frac{\tau_1 \leq \tau_2 \quad \forall v{:}\tau_1, [v/x]b_1 \mapsto^* \mathsf{true} \text{ implies } [v/x]b_2 \mapsto^* \mathsf{true}}{\{x{:}\tau_1 \mid b_1\} \leq \{x{:}\tau_2 \mid b_2\}}\ \mathsf{refine}$$

Figure 5.4: Subtyping

We translate casts with recursive types as follows. For each pair of compatible recursive types A and B, we generate a unique monitor name f and record its type $f : \{A \leftarrow B\}$ in a context Σ. The translation algorithm needs to take additional arguments, including Σ to generate and invoke the appropriate recursive process when needed. For instance, when generating the monitor process for $f : \{\mathsf{list} \leftarrow \mathsf{list}\}$, we follow the rule for translating internal choices. For $[\![\langle \mathsf{list} \Leftarrow \mathsf{list} \rangle^{\rho}]\!]_{y,x}$ we apply the cons case in the translation to get $y \leftarrow\ f\ \leftarrow x$.

5.4 Metatheory

We prove two formal properties of cast-based monitors: safety and transparency. We also prove preservation in the presence of well-typed casts.

Because of the expressiveness of our contracts, a general safety (or blame) theorem is difficult to achieve. However, for cast-based contracts, we can prove that a cast which enforces a subtyping relation, and the corresponding monitor, will not raise an alarm.

We first define our subtyping relation in Figure 5.4. In addition to the subtyping between refinement types, we also include label subtyping for our session types. A process that offers more external choices can always be used as a process that offers fewer external choices. Similarly, a process that offers fewer internal choices can always be used as a process that offers more internal choices (e.g., non-empty list can be used as a list). The subtyping rules for internal and external choices are drawn from work by Gay and Hole [15]. For recursive types, we directly examine their definitions. Therefore, our subtyping rules are co-inductively defined.

Our safety theorem guarantees that well-typed casts do not raise alarms. The key is to show that the monitor process generated from the translation algorithm in Figure 5.3 is well-typed under a typing relation which guarantees that no $\mathsf{abort}(l)$ state can be reached.

We refer to the type system presented thus far in the proposal as T, where monitors that may evaluate to $\mathsf{abort}(l)$ can be typed. We define a stronger type system S which consists of the rules in T with the exception of the abort rule and we replace the assert rule with the assert_strong rule. This new rule verifies that the condition b is true using the fact that the refinements are

stored in the context Ψ. The two type systems are summarized in Figure 5.7.

$\mathsf{list\_len} : \mathsf{int} \to \{\mathsf{list}[m] \leftarrow \mathsf{list}[n]\}$
$k \leftarrow \mathsf{list\_len}\ m \leftarrow l =$
  $\mathsf{case}\ l\ \mathsf{of}$
  $\mid \mathsf{nil} \Rightarrow \mathsf{assert}(count > 0)\ //check\ that\ k\ has\ enough\ elements$
    $k.\mathsf{nil}\ ;\ \mathsf{wait}\ l\ ;\ \mathsf{close}\ k$
  $\mid \mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l\ ;\ m = m - 1\ ;$
    $k.\mathsf{cons}\ ;\ \mathsf{send}\ k\ x\ ;\ k \leftarrow \mathsf{list\_len}\ m\ \leftarrow l$

Figure 5.5: List Refinement Monitor

$[\![\langle \mathsf{list}[m] \Leftarrow \mathsf{list}[n] \rangle^{\rho}]\!]_{k,l} = k \leftarrow \mathsf{list\_len}\ m \leftarrow l$
$\mathsf{list\_len} : \mathsf{int} \to \{\mathsf{list} \leftarrow \mathsf{list}\}$
$k \leftarrow \mathsf{list\_len}\ m \leftarrow l =$
  $\mathsf{case}\ l\ \mathsf{of}$
  $\mid \mathsf{nil} \Rightarrow \mathsf{assert}\ \rho\ (m \leq 0)\ ;\ k.\mathsf{nil}\ ;\ \mathsf{wait}\ l\ ;\ \mathsf{close}\ k$
  $\mid \mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l\ ;\ k.\mathsf{cons}\ ;\ \mathsf{send}\ k\ x\ ;\ k \leftarrow \mathsf{list\_len}\ (m-1) \leftarrow l$

Figure 5.6: List length monitor

We state the theorems below; the proof details may be found in Gommerstadt et al [17].

Theorem 2 (Refinement monitors are well-typed)**.**

1. $b : B \vdash_T [\![\langle A \Leftarrow B \rangle^{\rho}]\!]^{\Psi}_{a,b}$ *::* $(a : A)$.
2. *If* $B \leq A$, *then* $b : B \vdash_S [\![\langle A \Leftarrow B \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A)$.

Proof. The proof is by induction over the monitor translation rules. To prove (1) we use type system T. To prove (2) we use type system T and the sub-typing relation to show that (a) for the internal and external choice cases, no branches that include abort are generated; and (b) for the forall and exists cases, the assert never fails (i.e., the assert_strong rule applies). □

As a corollary, we can show that when executing in a well-typed context, a monitor process translated from a well-typed cast will never raise an alarm.

Corollary 1 (Well-typed casts cannot raise alarms)**.** $\vdash \mathcal{C} :: (b : B)$ *and* $B \leq A$ *implies* $\mathcal{C}, \mathsf{proc}(a, [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}) \not\longrightarrow^* \mathsf{abort}(\rho)$.

Proof. By Theorem 2 we have that $\Psi\ ;\ b : B \vdash_S [\![\langle A \Leftarrow B \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A)$. Type system S does not allow aborts, so $\mathcal{C}, \mathsf{proc}(a, [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}) \not\longrightarrow^* \mathsf{abort}(\rho)$. □

Theorem 3 (Casts are transparent)**.**
$b : B \vdash \mathsf{proc}(b, a \leftarrow b) \sim \mathsf{proc}(a, [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}) :: (a : A)$.

Proof. We just need to show that the translated process passes the partial identity checks. We can show this by induction over the translation rules and by applying the rules in Gommerstadt et al [16]. □

We now present the configuration typing in order to state the preservation theorem. We assume that the comma operator is associative with $\cdot$ as the unit.

$$\mathcal{C} = \cdot \mid \mathsf{proc}(c, P) \mid \mathsf{msg}(c, P) \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{abort}(l)$$

$$\frac{}{\Vdash \cdot} \quad \frac{\Vdash \mathcal{C}_1 \quad \Vdash \mathcal{C}_2}{\Vdash \mathcal{C}_1, \mathcal{C}_2} \quad \frac{\Delta \vdash P :: (c : A)}{\Vdash \mathsf{proc}(c, P)} \quad \frac{\Delta \vdash P :: (c : A)}{\Vdash \mathsf{msg}(c, P)}$$

Theorem 4 (Subtyping-Preservation)**.** *If* $\Delta_1 \Vdash \mathcal{C} : \Delta_2$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$ *then* $\Delta_1 \Vdash \mathcal{C}' : \Delta_2$.

Proof. We make use of typed semantics which augment the message processes with channel types and casts as appropriate. We prove the theorem by examining the semantics, and invoking the inversion and subtype-substitution lemmas. □

5.5 Related Work

Many of the contracts studied in the context of the lambda calculus [1, 10, 11, 14, 24, 37] are based on refinement types. Our contracts are able to encode refinement-based contracts. Our safety theorem supports Walder and Findler's [38] claim that the less-precisely typed code is always to blame. When we cast from a type to its supertype, the cast can never be at fault, and remains well-typed. A cast can only be at fault when coercing a less-precise type to a more-precise type. In this situation, we generate a monitor to validate the cast.

Recently, gradual typing for two-party session-type systems has been developed [21, 32]. Even though it is a different formalism, the way untyped processes are gradually typed at runtime resembles how we monitor type casts. Because of dynamic session types, their system has to keep track of the linear use of channels, which is not needed for our monitors.

5.6 Proposed Work: Length Refinements for Lists

We propose to extend our type refinements to encompass length refinements on lists. For example, $\mathsf{list}[5]$ would represent the type for a list of five integers. In Figure 5.6, we show how our translation could be extended to handle casts between lists of different lengths.

Both System T and S

$$\frac{}{\Psi\ ;\ b : A \vdash a \leftarrow b :: (a : A)}\ \mathsf{id} \qquad \frac{\Psi\ ;\ \Delta \vdash P :: (x : A) \quad x : A, \Delta' \vdash Q :: (c : C)}{\Psi\ ;\ \Delta, \Delta' \vdash x{:}A \leftarrow P\ ;\ Q :: (c : C)}\ \mathsf{cut}$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : A^+)}{\Psi\ ;\ \Delta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \uparrow A^+)}\ \uparrow R \qquad \frac{\Psi\ ;\ \Delta, c : A^+ \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \uparrow A^+ \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ Q :: (d : D)}\ \uparrow L$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : A^-)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ P :: (c : \downarrow A^-)}\ \downarrow R \qquad \frac{\Psi\ ;\ \Delta, c : A^- \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \downarrow A^- \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \downarrow L$$

$$\frac{}{\cdot \vdash \mathsf{close}\ c :: (c : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Psi; \Delta \vdash Q :: (d : D)}{\Psi; \Delta, c : \mathbf{1} \vdash \mathsf{wait}\ c\ ;\ Q :: (d : D)}\ \mathbf{1}L$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : B)}{\Psi\ ;\ \Delta, a : A \vdash \mathsf{send}\ c\ \ a\ ;\ P :: (c : A \otimes B)}\ \otimes R \qquad \frac{\Psi; \Delta, x : A, c : B \vdash Q :: (d : D)}{\Psi; \Delta, c : A \otimes B \vdash x \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \otimes L$$

$$\frac{\Psi; \Delta, x : A \vdash P :: (c : B)}{\Psi; \Delta \vdash x \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : A \multimap B)}\ \multimap R \qquad \frac{\Psi\ ;\ \Delta, c : B \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, a : A, c : A \multimap B \vdash \mathsf{send}\ c\ a\ ;\ Q :: (d : D)}\ \multimap L$$

$$\frac{\Psi; \Delta \vdash P_\ell :: (c : A_\ell) \quad \text{for every } \ell \in L}{\Psi; \Delta \vdash \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (c : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R \qquad \frac{k \in L \quad \Psi; \Delta, c : A_k \vdash Q :: (d : D)}{\Psi; \Delta, c : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c.k\ ;\ Q :: (d : D)}\ \&L$$

$$\frac{k \in L \quad \Psi; \Delta \vdash P :: (c : A_k)}{\Psi; \Delta \vdash c.k\ ;\ P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R \qquad \frac{\Psi; \Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Psi; \Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)}\ \oplus L$$

$$\frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta \vdash P :: (c : [v/n]A)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ v\ ;\ P :: (c : \exists n{:}\tau.\ A)}\ \exists R \qquad \frac{\Psi, n{:}\tau\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \exists n{:}\tau.\ A \vdash n \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \exists L$$

$$\frac{\Psi, n{:}\tau\ ;\ \Delta \vdash P :: (c : A)}{\Psi\ ;\ \Delta \vdash n \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \forall n{:}\tau.\ A)}\ \forall R \qquad \frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \forall n{:}\tau.\ A \vdash \mathsf{send}\ c\ v\ ;\ Q :: (d : D)}\ \forall L$$

$$\frac{\Psi \vdash v : \tau' \quad \Psi, x : \tau\ ;\ \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'}{\Psi\ ;\ \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau' \rangle^\rho\ v\ ;\ Q :: (c : C)}\ \mathsf{val\_cast} \qquad \frac{A \sim B}{\Psi, b\ ;\ b : B \vdash a \leftarrow \langle A \Leftarrow B \rangle^\rho\ b :: (a : A)}\ \mathsf{id\_cast}$$

System T only

$$\frac{\Psi \vdash b : \mathsf{bool} \quad \Psi\ ;\ \Delta \vdash Q :: (x : A)}{\Psi\ ;\ \Delta \vdash \mathsf{assert}\ \rho\ b; Q :: (x : A)}\ \mathsf{assert} \qquad \frac{}{\Psi\ ;\ \Delta \vdash \mathsf{abort}\ \rho :: (x : A)}\ \mathsf{abort}$$

System S only

$$\frac{\Psi \vDash b\ \mathsf{true} \quad \Psi\ ;\ \Delta \vdash Q :: (x : A)}{\Psi\ ;\ \Delta \vdash \mathsf{assert}\ \rho\ b; Q :: (x : A)}\ \mathsf{assert\_strong}$$

Figure 5.7: Typing process expressions

Chapter 6

Proposed Work: Dependent Types as Contracts

While Chapter 5 discussed monitoring type refinements, a special case of dependent types, monitoring arbitrary dependent types presents unique challenges. Dependent type theories, such as Coq, Agda, and Nuprl, encode dependent types by integrating programs and proof objects. These proof objects must be generated, and sent through the system. Transmitting proof objects requires significant infrastructure [26], so a lightweight approach to dependent contract checking is desirable. The proof objects must then be verified which reduces to type checking. In certain cases, proof objects encode constraints that are simple to validate, such as the fact that an integer is positive. In these situations, it is unneccessary to send a proof object. The concept of *proof irrelevance* [28] captures the idea that some proofs play no computational role in the program. If the proof object is irrelevant, we can avoid sending the proof object and check the truth value of the proposition directly. Unfortunately, even if the proof object is irrelevant, many constraints, such as the fact that a function is monotonic, are hard to validate. In these situations, dynamic monitors, which can check the conditions encoded by the proof objects, are necessary. We aim to provide a more lightweight approach to dependent contract checking by generating partial identity monitors for dependent session types.

In this chapter, we encode dependent session types with proof objects [30] which allows processes to remain well-typed with respect to the non-dependent type system (Section 2.1). We then provide examples of contracts expressed as dependent types, and examine partial identity monitors (discussed in Section 4) that are able to check those contracts. We propose a translation that will transform session types containing proof objects into monitors. These monitors will validate the constraints encoded by the proof objects at runtime. We conclude the chapter by identifying properties of the translation that we plan to prove and surveying related work.

6.1 Surface Language

The core element of the surface language is an extension to the data layer which allows the session-typed processes to send and receive proof objects. The data values, including proof objects, are taken from the underlying functional layer, which is dependently typed. The proof

objects are not part of the session-typed language. The session types that interact with the data layer are $\forall n : \tau.A$ and $\exists n : \tau.A$. We extend the data layer as follows where p is a proposition.

$$\tau ::= ... \mid p \mid \Pi x : \tau.\sigma \mid \Sigma x : \tau.\sigma$$

The simplest proof object can be thought of as a predicate function, while more complicated proof objects will make use of the existential (Σ) and universal (Π) quantifiers, as shown in the next section.

6.2 Examples

We first consider a simple dependent type representing an interaction where a process receives an integer, sends an integer and then terminates. We want to monitor that both integers are positive which is encoded by the proof objects p and q. We show how to write a monitoring process for this contract in Figure 6.1. Essentially, each proof object is translated into an assertion.

$$\mathsf{simple\_t} : \forall n : \mathsf{int}.\forall p : [n > 0].\exists y : \mathsf{int}.\exists q : [y > 0].\mathbf{1}$$

$\mathsf{simple\_mon} : \{\mathsf{simple\_t} \leftarrow \mathsf{simple\_t}\}$
$c' \leftarrow \mathsf{simple\_mon} \ \leftarrow c =$
  $x \leftarrow \mathsf{recv}\ c$; $\mathsf{assert}\ \rho_1\ (x > 0)$; $\mathsf{send}\ c'\ x$;
  $z \leftarrow \mathsf{recv}\ c'$; $\mathsf{assert}\ \rho_2\ (z > 0)$; $\mathsf{send}\ c\ z$;
  $\mathsf{wait}\ c$; $\mathsf{close}\ c'$;

Figure 6.1: Simple dependent monitor

We now examine a monitor to check that a list is in ascending order. We can already write this monitor by using techniques from Section 4. We show the non-dependent version in Figure 6.2. The non-dependent version makes use of an integer argument to store a bound on the list elements. If the list is empty, there is no bound to check, so no contract failure can happen. If the list is nonempty, we check if the received element is greater or equal to the bound, and recurse with the updated bound. If the received element is less than the bound, then the list is not in ascending order, so we abort with a contract failure. Because every list containing one element is in ascending order, we initially set the bound to be negative infinity.

$\mathsf{asc\_mon1} : \mathsf{int} \rightarrow \{\mathsf{list} \leftarrow \mathsf{list}\}$
$m \leftarrow \mathsf{asc\_mon1}\ \mathit{bound} \leftarrow n =$
  $\mathsf{case}\ n\ \mathsf{of}$
  $\mid \mathsf{nil} \Rightarrow m.\mathsf{nil}$; $\mathsf{wait}\ n$; $\mathsf{close}\ m$
  $\mid \mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ n$; $\mathsf{assert}\ \rho\ (x \geq a)$; $m.\mathsf{cons}$; $\mathsf{send}\ m\ x$; $m \leftarrow \mathsf{asc\_mon1}\ x \leftarrow n$

Figure 6.2: Ascending monitor version 1

We can also express the requirement that a list be in ascending order by the following dependent type where the list is indexed by a lower bound.

$$\mathsf{list}(x) = \oplus\{\mathsf{nil} : 1; \mathsf{cons} : \exists y : \mathsf{int}.\exists p : [y \geq x].\mathsf{list}(y)\}$$

A monitor for this type is shown in Figure 6.3. This monitor stores the head of the list as its argument. When the monitor receives an element, it checks if the received element is greater or equal to its argument, and recurses with the updated argument. If the received element is less than the argument, then the list is not in ascending order, so the monitor aborts with a contract failure.

$\mathsf{asc\_mon2} : \mathsf{int} \to \{\forall x : \mathsf{int}.\mathsf{list}(x) \leftarrow \mathsf{list}(x)\}$
$m \leftarrow \mathsf{asc\_mon2}\ x\ \leftarrow n =$
  $\mathsf{case}\ n\ \mathsf{of}$
  $\mid \mathsf{nil} \Rightarrow m.\mathsf{nil}\ ;\ \mathsf{wait}\ n\ ;\ \mathsf{close}\ m$
  $\mid \mathsf{cons} \Rightarrow y \leftarrow \mathsf{recv}\ n\ ;\ \mathsf{assert}\ \rho\ (y \geq x)\ ;\ m.\mathsf{cons}\ ;\ \mathsf{send}\ m\ y\ ;\ m \leftarrow \mathsf{asc\_mon2}\ y\ \leftarrow b\ ;$

Figure 6.3: Ascending monitor version 2

Finally, we consider a dependent type that makes use of the Π constructor. This type models a process that receives a function on integers, a proof that the function is increasing, and emits a fixed point of that function if it exists.

$$\mathsf{depend\_t} : \forall f : \mathsf{int} \to \mathsf{int}.\forall p : [\Pi x : \mathsf{int}.f(x) \geq x].\exists y : \mathsf{int}.\exists q : [y = f(y)].1$$

In order to generate a monitor for the depend_t type, we have to handle both proof objects present in the type. We have two choices for how to handle the function f. We can either integrate functions into our session-typed language or we can interpret functions as processes. We anticipate choosing the former approach. In this case, a separate monitor will be responsible for monitoring that the function f is increasing. This monitor will not be a session-typed process, but rather a functional program. Checking that y is the fixpoint of f can be done directly with an assertion. The partial identity monitor for the depend_t type will be responsible for combining both the functional program and the assertion to handle all of the proof objects.

Providing a mechanism for monitoring contracts with Π and Σ constructors is the core challenge of the proposed work.

6.3 Translation to Monitors

At runtime, a dependent session-type containing proof objects will be translated into a monitoring process which will validate the proof objects. A proof object p will be implemented as an assertion or a functional program.

The translation is of the form: $[\![A]\!]_{a,b} = P$, where A is a dependent session-type containing proof objects. We assume channels a and b are the offering channel and monitoring channel (respectively) for the resulting monitoring process P. The translation T will be defined inductively over the structure of the types and will encompass the examples described in the previous section.

6.4 Metatheory

We will aim to prove the following monitor properties:

- Dependent monitors are well-typed (safety)
- Transparency
- Preservation

6.5 Related Work

Because dependent types subsume refinement types, the related work described in Section 5.5 is also relevant here. Greenberg et al. [18] survey the the literature on dependent contract checking, which focuses on dependent functions, and compares the technical approaches. Most relevant to our work is that of Ou et al. [27] who have developed a language where simply-typed and dependently-typed code is integrated by using coersions. Each coersion is treated as a contract and checked dynamically. Their coercions resemble the proof objects in our system and we will investigate the connections between them.

Recently, Toninho and Yoshida [35] have presented data-dependent session types, which are a version of dependent types that integrate dependent functions and session-typed processes. Their type theory can express protocols where the choice of the next communication action can depend on specific values of the received data. As an example, consider the below type:

$$\forall x : \mathsf{int}.\mathsf{if}\ (x > 0)(\exists .y : \mathsf{int}.\mathbf{1})(\mathbf{1})$$

This type represents a process that receives the integer x and checks whether it is positive. It it is, the process sends an integer y and then terminates. If not, the process terminates. We will investigate how to monitor contracts expressed by data-dependent session types and whether our translation can generate these monitors.

Chapter 7

Proposed Work: Miscellaneous Monitoring Extensions

7.1 Integrating Monitoring and Resource Analysis

While we are able to use session-typed monitors to check various classes of contracts, we do not have a way of reasoning about the resources these monitors consume. Recently, Das et al. have proposed resource-aware session types to statically analyze resource usage for concurrent message-passing programs. They initially present a type system allowing processes and messages to send potential to analyze the total work performed by a system [9]. In subsequent work, they add temporal modalities to session types to analyze the parallel complexity of a system [8]. In both of these models, the programmer is able to provide the system with a cost model to fit the particular interaction. These resource models can be applied to our session-typed partial identity monitors (described in Section 4) to determine static bounds on monitor cost. Ankush Das, Adrian Francalanza, and I have started a project to combine partial identity monitors and resource aware session-types. Ongoing work on this project is part of the proposed work for this thesis.

7.2 Beyond Partial Identity Monitors: Unrestricted Channels

As an example, let us consider how to create a stream that consists of the bitwise logical disjunction (denoted $\vee$) of two streams of bits, implemented on channels x and y. The first method involves defining a process that examines each corresponding bit of x and y and then computes the resulting bit appropriately to output on channel z. The second method uses the negation of logical or (denoted $\downarrow$) and the following fact from propositional logic:

$$x \vee y = (x \downarrow y) \downarrow (x \downarrow y)$$

We note that the channels x and y cannot be linear in order to use the above property to implement *or* using nor, because each of them is used twice. Let the standard implementation be called or_std, and the one using nor be called or_nor. The implementation of or_nor is shown in Figure 7.1.

We would like to enforce the contract that the or_nor process produces the same result as or_std process does. A monitor for this contract would need access to the channels x and y in order to pass those channels along to the or_std process. The monitor will also need to make use of a process to check whether two streams of bits have the same bits (referred to as eq). The code for this monitor is shown in Figure 7.1.

$z \leftarrow \mathsf{or\_nor} \leftarrow x, y =$
  $u \leftarrow \mathsf{nor} \leftarrow x, y$;
  $z \leftarrow \mathsf{nor} \leftarrow u, u$

$z \leftarrow \mathsf{or\_mon} \leftarrow x, y, z' =$
  $z'' \leftarrow \mathsf{or\_1} \leftarrow x, y$;
  $z \leftarrow \mathsf{eq} \leftarrow z, z''$

Figure 7.1: Or/Nor monitor

Unfortunately, this monitor does not meet our criteria for a partial identity monitor because it takes multiple channels as an argument, even though two of them are non-linear. The goal of this proposed work is to investigate whether we can relax the partial identity rules to allow passing non-linear channels as arguments to monitoring processes. This relaxation would allow us to model another class of contracts, but would necessitate verifying that the changes do not jeopardize the transparency property of our monitors.

Chapter 8

Conclusion

8.1 Timeline

Month(s)	**Tasks**
Dec	Proposal
Jan	Finish proofs from Section 3 for the *sharpened* system Finish proofs from Section 3 for the *strong* system Submit journal paper proposed in Section 3 Collaborate with Ankush and Adrian on monitor resource analysis from Section 7.1
Feb	Define translation in Section 6 Work on Section 6 metatheory, attempt to prove stated properties Collaborate with Ankush and Adrian on monitor resource analysis from Section 7.1
Mar	Continue working on Section 6 metatheory, submit to ICFP if ready Collaborate with Ankush and Adrian on monitor resource analysis from Section 7.1
Apr	Continue working on Section 6 metatheory, submit to CONCUR if ready Collaborate with Ankush and Adrian on monitor resource analysis from Section 7.1
May - June	Write thesis document
July	Send thesis to committee and prepare defense talk Submit work from Section 6 or 7 to POPL
Aug	Defend thesis

Bibliography

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, 2011. doi: 10.1145/1570506.1570507. URL https://doi.acm.org/10.1145/1570506.1570507. 2.2, 5.5

[2] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proc. ACM Program. Lang.*, 1(ICFP):37:1–37:29, August 2017. ISSN 2475-1421. doi: 10.1145/3110281. URL https://doi.acm.org/10.1145/3110281. 2.1

[3] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *Formal Techniques for Distributed Systems (FMOODS 2013)*, 2013. 3.4

[4] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory (CONCUR 2010)*, 2010. doi: 10.1007/978-3-642-15375-4_16. URL https://dx.doi.org/10.1007/978-3-642-15375-4_16. 2.1

[5] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. Special Issue on Behavioural Types. 2.1

[6] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, 2009. doi: 10.1016/j.ic.2008.11.006. URL https://dx.doi.org/10.1016/j.ic.2008.11.006. 2.1

[7] Tzu-Chen Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *6th International Symposium on Trustworthy Global Computing (TGC 2011)*, pages 25–45, Aachen, Germany, June 2012. Springer LNCS 7173. 3.4

[8] Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. *Proc. ACM Program. Lang.*, 2(ICFP):91:1–91:30, July 2018. ISSN 2475-1421. doi: 10.1145/3236786. URL http://doi.acm.org/10.1145/3236786. 7.1

[9] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 305–314, 2018. doi: 10.1145/

3209108.3209146. URL http://doi.acm.org/10.1145/3209108.3209146. 7.1

[10] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 215–226, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926410. URL https://doi.acm.org/10.1145/1926385.1926410. 2.2, 5.5

[11] Christos Dimoulas, Sam T. Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *21st European Conference on Programming Languages and Systems (ESOP 2012)*, 2012. doi: 10.1007/978-3-642-28869-2_11. URL https://dx.doi.org/10.1007/978-3-642-28869-2_11. 2.2, 5.5

[12] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In *16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, 2011. doi: 10.1145/2034773.2034800. URL https://doi.acm.org/10.1145/2034773.2034800. 4.4

[13] Luminous Fennell and Peter Thiemann. The blame theorem for a linear lambda calculus with type dynamic. In *13th International Symposium on Trends in Functional Programming (TFP 2012)*, 2012. 2.2

[14] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581484. URL https://doi.acm.org/10.1145/581478.581484. 1, 2.2, 5.5

[15] Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005. doi: 10.1007/s00236-005-0177-z. URL https://dx.doi.org/10.1007/s00236-005-0177-z. 2.1, 5.4

[16] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. Session-typed concurrent contracts. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 771–798, 2018. doi: 10.1007/978-3-319-89884-1_27. URL https://doi.org/10.1007/978-3-319-89884-1_27. 4.1, 4.3, 1, 5.1, 5.4

[17] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. Session-typed concurrent contracts. Technical Report CMU-CyLab-17-004, CyLab, Carnegie Mellon University, February 2018. 5.4

[18] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 353–364, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706341. URL http://doi.acm.org/10.1145/1706299.1706341. 6.5

[19] Dennis Griffith. *Polarized Substructural Session Types*. PhD thesis, University of Illinois at Urbana-Champaign, April 2016. 2.1

[20] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory (CONCUR 1993)*, 1993. 2.1

[21] Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *Proc. ACM Program. Lang.*, 1(ICFP):38:1–38:28, August 2017. ISSN 2475-1421. doi: 10.1145/3110282. URL https:/doi.acm.org/10.1145/3110282. 5.5

[22] Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. Technical Report CMU-CyLab-15-004, CyLab, Carnegie Mellon University, November 2015. 3.3

[23] Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 582–594, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837662. URL https://doi.acm.org/10.1145/2837614.2837662. 3.1

[24] Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In *20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*, 2015. doi: 10.1145/2784731.2784737. URL https://doi.acm.org/10.1145/2784731.2784737. 2.2, 5.5

[25] Hernán Melgratti and Luca Padovani. Chaperone contracts for higher-order sessions. *Proc. ACM Program. Lang.*, 1(ICFP):35:1–35:29, August 2017. ISSN 2475-1421. doi: 10.1145/3110279. URL https://doi.acm.org/10.1145/3110279. 4.4

[26] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi: 10.1145/263699.263712. URL http://doi.acm.org/10.1145/263699.263712. 1, 6

[27] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, Boston, MA, 2004. Springer US. ISBN 978-1-4020-8141-5. 6.5

[28] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, pages 221–, Washington, DC, USA, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=871816.871845. 6

[29] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, 2015. doi: 10.1007/978-3-662-46678-0_1. URL https://doi.acm.org/10.1007/978-3-662-46678-0_1. Invited talk. 2.1, 4.2

[30] Frank Pfenning, Luís Caires, and Bernardo Toninho. Proof-carrying code in a session-typed

process calculus. In *1st International Conference on Certified Programs and Proofs (CPP 2011)*, 2011. 6

[31] Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and Coercion: Together Again for the First Time. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, 2015. 2.2

[32] Peter Thiemann. Session Types with Gradual Typing. In *9th International Symposium on Trustworthy Global Computing (TGC 2014)*, 2014. doi: 10.1007/978-3-662-45917-1_10. URL https://dx.doi.org/10.1007/978-3-662-45917-1_10. 5.5

[33] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 462–475, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.4230/LIPIcs.ECOOP.2016.9. URL https:/acm.doi.org/10.4230/LIPIcs.ECOOP.2016.9. 4.2

[34] Bernardo Toninho. *A Logical Foundation for Session-based Concurrent Computation*. PhD thesis, Carnegie Mellon University and New University of Lisbon, 2015. 2.1

[35] Bernardo Toninho and Nobuko Yoshida. Depending on session-typed processes. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 128–145, 2018. doi: 10.1007/978-3-319-89366-2_7. URL https://doi.org/10.1007/978-3-319-89366-2_7. 6.5

[36] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *22nd European Symposium on Programming (ESOP 2013)*, 2013. doi: 10.1007/978-3-642-37036-6_20. URL https://dx.doi.org/10.1007/978-3-642-37036-6_20. 1, 2.1

[37] Philip Wadler. A Complement to Blame. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, 2015. doi: 10.4230/LIPIcs.SNAPL.2015.309. URL https://doi.acm.org/10.4230/LIPIcs.SNAPL.2015.309. 2.2, 5.5

[38] Philip Wadler and Robert B. Findler. Well-Typed Programs Can't Be Blamed. In *18th European Symposium on Programming Languages and Systems (ESOP 2009)*, 2009. doi: 10.1007/978-3-642-00590-9_1. URL https://dx.doi.org/10.1007/978-3-642-00590-9_1. 2.2, 3.4, 5, 5.5

[39] Lucas Waye, Stephen Chong, and Christos Dimoulas. Whip: Higher-order contracts for modern services. *Proc. ACM Program. Lang.*, 1(ICFP):36:1–36:28, August 2017. ISSN 2475-1421. URL https://doi.acm.org/10.1145/3110280. 3.4, 4.4