On Session Typed Contracts for Imperative Languages

Chuta Sano (andrew id: csano)

November 26, 2019

Abstract

Session types prescribe the protocols for communication between concurrently executing processes. Following discoveries of the correspondence between intuitionistic linear logic and linear session types, there have been many related works ranging from practical implementations to theoretical extensions. In particular, linear session types, which assume a strong condition that there is only one client for a session, have been extended to shared session types, which introduce semantics for multiple clients. This extension was subsequently implemented in an imperative setting in the language Concurrent C0. In another direction, contracts, which are well-studied constructs in ordinary type systems, have been extended to monitors, which are contracts for linear session types, in the usual functional setting. In this work, we formalize an imperative programming language based on previous work which implements both linear and shared session typed channels and adapt the monitors to the imperative setting. We further extend the notion of monitoring to shared session types and introduce its semantics. Finally, we introduce several case studies of linear and shared monitors.

1 Introduction

1.1 C0 and Contracts

C0 is a C-like language used in CMU's Principles of Imperative Computation, a typical class for first year undergraduate computer science students. Aside from being a simpler subset of C, C0 also provides type and memory safety. Furthermore, C0 supports a rich variety of contracts, which help to enforce a disciplined style of programming [Pfe10; Arn10].

To give an intuitive understanding of contracts, we first consider one of the most common and primitive examples of contracts: assertions. In C0, an assertion is expressed as a statement of form `//@assert e` for some boolean expression e, where the semantics of this statement is that if e evalutes to true, the program continues, making the statement effectively a no-op, and on the other hand if e evalutes to false, then the program aborts with an error message.

Assertions are widely available across various imperative programming languages such as C, C++, and Java, and despite minor variances in syntax and failure consequences, all asserts can be suppressed during compile time, allowing the same source code to produce a slower debug build for developers and a faster assertion-free production build for releases.

Since assertions can be enabled and disabled based on compiler settings, the actual presense of assertions should not affect the state of the program assuming the particular assertion is satisfied. We do mention that certain languages do not explictly check this requirement, such as C, where for example, one can technically write effectful assertions such as `assert(x++ > 0)`. This is considered bad practice and is heavily discouraged for good reason. In C0 however, this requirement is enforced by the compiler through a pass called the purity check.

More generally, contracts can be thought of as runtime checks and tests that programmers can write which are effectless given that the tests pass. Contracts should be suppressable, meaning that a compiler should be able to transform a program with contracts into an equivalent program without contracts (again, assuming that these runtime tests pass). In this paper, we capture the recurring notion of "assuming that the tests pass" as "partial."

With this terminology, we can succinctly describe contracts as runtime tests that when part of a program, are partially equivalent to the same program with contracts disabled. An assertion is then a subset of a contract; it is a statement with a predicate which when executed is partially equivalent to a no-op. Since contracts help catch unintended program states and immediately halt the program on failure, they are not only useful in a teaching environment to train good coding discipline but also in industry, where code is updated frequently on large projects.

We also would like to point out that there are more considerations for contracts. For example, a very important topic regarding contracts is the content of the error messages on failure. Concepts like blame assignment, tracing, etc. are also very important aspects of contracts, but due to the preliminary nature of this work, we leave out many of these components and focus only on the fundamental idea behind contracts.

Including the aforementioned assertions, contracts in C0 come in four flavors: pre-conditions, post-conditions, loop invariants, and asserts. Pre-conditions (@requires) state requirements at function entry, for example properties of arguments, whereas post-conditions (@ensures) state requirements after a function call, for example properties of return values or invariants of parameters that are passed by reference. Loop invariants are assertions that must be true before the exit condition. See Listing 1 for an example of the four contracts.

Listing 1: An example of a function in C0 with contracts [Pfe10].

```
int log(int x)
//@requires x >= 1;
//@ensures \result >= 0;
```

```
 4 //@ensures (1 << \result) <= x;
 5 { int r = 0;
 6   int y = x;
 7   while (y > 1)
 8     //@loop_invariant y >= 1 && r >= 0;
 9     //@loop_invariant y * (1 << r) <= x;
10     {
11       y = y / 2;
12       //@assert r < x;
13       r = r + 1;
14     }
15   return r;
16 }
```

The pre-condition at line 2 requires that $x >= 1$ on function call. The post-condition at line 3 demonstrates the use of the special variable $\backslash result$, which allow post-condition contracts to reference the return value of the function. The post-condition at line 4 shows a usage of a parameter; this is only allowed because x is not mutated within the function definition (hence we copy the value of x to y at line 6). Inside the function body, two loop invariants are demonstrated at lines 8 and 9, and an assert is demonstrated at line 12.

1.2 Linear Session Types

Linear session types, initially coined as "session types," prescribe bidirectional communication protocols between concurrent processes and guarantee strong properties like deadlock freedom [Hon93; HVK98]. The linearity condition requires that a given session must have exactly one client/user and server/provider. This type system was subsequently shown to correspond to intuitionistic linear logic [CP10].

We briefly introduce a modified portion of Honda's formulation to provide an intuition for linear session types. A more complete and formal formulation is provided in section 2. We first begin with some type σ. Then we introduce the input and output type constructors $?\sigma$ and $!\sigma$ which operationally means to receieve and send some value of type σ respectively. These input and output types are called "action types." These action types can then be composed [1] into a sequence $a_1; a_2$, with the meaning being that in this type, an action a_1 is performed and then an action a_2 is performed. A sequence of actions is still an action, meaning arbitrary long chains of actions can be created.

For example, consider a server that when given a natural number, returns a single prime factor of that natural number. From the server's perspective, the protocol can be written $?\mathbf{int}; !\mathbf{int}$. On the other hand, from a client's perspective, the actions are "flipped": $!\mathbf{int}; ?\mathbf{int}$, since a client, to satisfy the server's protocol, must first send a natural number and then receive the resultant natural number (presumed to be some prime factor).

[1]The original formulation by Honda does not distinguish action types and ordinary types for type constructors. We present this distinguishment to better transition to our typesystem.

We next introduce branching constructors $a_1 \ \& \ a_2$ and $a_1 \ \oplus \ a_2$: $a_1 \ \& \ a_2$ offers two alternative actions while $a_1 \ \oplus \ a_2$ allows for the selection of one of two actions.

For example, consider a server that offers a random number. Perhaps it allows a user to select between an **int** and **bool**, both cases providing a random value within the domain of the respective type. In this case, the server offers a choice between two actions !**int** and !**bool** since after the choice has been made by a user, it must send a value. The server's protocol can therefore be denoted as !**int** & !**bool**. On the other hand, from the client's perspective, it must choose between the two actions ?**int** and ?**bool** since we must receive a value after making some selection. The client's protocol can therefore be denoted as ?**int** $\oplus$?**bool**.

As shown, each session type when viewed from the server's perspective gives rise to a dual session type when viewed from the client's perspective. By convention, we choose to view session types from the server's perspective, meaning when typechecking the client's interactions, we implicitly infer the dual of the session type. We also would like to point out that due to bidirection, the direction of the message is in no way related to whether the process using the channel is providing the channel (server) or using the channel (client).

From an operational standpoint, similar to how a value of an ordinary type can be held by a variable, a single session adhering to some session type (a protocol) can be held by a channel. However, the operations one can perform on channels are not at all similar to ordinary variables since channels do not hold values; they instead hold an avenue of communication to some endpoint. Although expanded later, the informal idea is that we can either send to a channel or receive from a channel. As described so far, we can send or receive values or choices, but we have not covered recursive session types nor any notion of termination of a session.

1.3 Shared Session Types

Although linear session types can express many protocols, the strong assumption of linearity means that we are not able to model many other situations. We recall that what made linear session types "linear" was the assumption that a given channel consists of exactly one client and one provider (server). A shared session type is a weakening of that assumption; a given shared channel consists of potentially multiple clients but only one provider.

One approach to implement this extension was presented by Caires [CP10]. To motivate this construction, we start by a simple scenario. Consider a server that provides a single random integer. In the context of linear session types, the protocol can then be informally expressed as !**int** (recall that we express session types from the provider/server's perspective). However, it's desirable for such a service to be available to multiple users with the intended use case being that multiple processes $P_1, \ldots, P_n$ can independently receive some random integer from the service. A more practical example is a typical web server, which certainly would like to serve multiple clients.

Caires resolved this problem by introducing the copy semantics shared session type. In the random number example, the protocol would instead be represented as some copiable session type: **copiable**(!**int**). Unlike the linear channels, a channel of this session type can have multiple clients. The idea then is that a client can then transform a copiable session type into its underlying linear session type and then proceed as if a linear session. From the server's point of view, it creates a copy of itself as many times as there are clients where each copy operates independently and act linearly.

Another approach by Balzer et al. implemented another semantics for this idea of a channel serving multiple clients: manifest sharing [BP17]. This new semantics of shared session types was motivated by the inability for the copy semantics version to implement shared resources or retain state; since copy semantics merely replicate a provider, the sessions across different clients are independent of each other, which cannot express mutable resources (if a client writes to a copied provider, then that information cannot be propagated to different copies).

The intuition behind manifest shared session types is that at any given time, only one client can acquire some shared channel. The client that succesfully acquired the shared channel must then at some point release the channel, allowing other clients waiting for the shared channel to acquire the channel. In short, a client must follow an acquire, linear, then release pattern. On the other hand, a server must accept an acquire request, act linearly, and then wait for its client to detach itself from the channel.

Since multiple clients can be waiting to acquire the same shared channel, any release must return the shared channel to its original type. Balzer et al. names this property "equi-synchronizing." This version of shared session types will be formalized in detail in section 2.

We summarize the three forms of session types: linear, copy shared, and manifest shared in figure 1.

1.4 Monitors

An addition of channels and session types to a language means that boolean predicates, which we saw as seemingly sufficient, cannot express meaningful tests for a session since channels do not hold any notion of "values." If for example, we listen alongside a channel to receive some value inside a boolean predicate, then we modify the program since an equivalent program with these hypothetical contracts suppressed will no longer receive, making the particular process no longer adhering to the protocol and therefore failing typechecking.

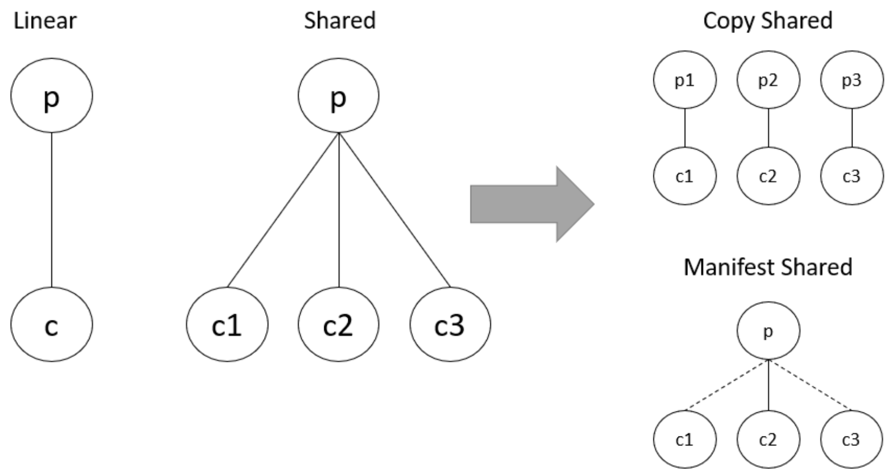

Figure 1: Left: Linear sessions have exactly one provider and client. Middle: The intended semantics of shared sessions require one provider but allow multiple clients ($c1, c2, c3$). Right: the two implementations of shared channels as described in this section. The dotted lines in manifest sharing indicate a blocked channel waiting to acquire. In the figure, $c2$ currently has exclusive access to p and $c1, c3$ are blocked waiting for access to p until $c2$ releases.

Instead, contracts for channels, or monitors, are formulated as intermediary processes that attach to a channel and listen along the channel [GJP18]. That is, given some session from a process P to another process Q, a monitor M must attach itself inbetween the two processes, replacing the session from P to Q with two sessions from P to M and M to Q. Thus, M effectively acts as a middleman that can listen along the communication and raise errors as necessary.

The requirement of partial equivalence under suppression or "no effect" can then be captured by imposing an additional requirement on the monitor; it must faithfully pass all messages it receives, unless a contract is violated. This restriction is called partial identity and is later developed in the context of an imperative language. Therefore, session typed contracts, or monitors, are just partial identity processes.

1.5 Concurrent C0

Concurrent C0, or CC0, was later developed which added inter-process asynchronous communication over linear session-typed channels to C0 [WPP17; GV10]. Shared session-types were then implemented over CC0 based on the aforementioned manifest sharing approach [BP17]. Therefore in this paper, unless explicitly mentioned, we refer to manifest sharing as "sharing" (similarly, manifest shared session types as "shared session types").

1.6 Problem Statement

Although CC0 introduced new features for the language, it does not support monitors, which are essentially contracts for session types. This causes a disconnect between the style of C0 and CC0, and although monitors for linear session types had been proposed in a functional setting [GJP18], the proposed system does not easily translate to the imperative setting of CC0. Furthermore, there is no known work of a corresponding system of monitors for shared session types.

Thus, our thesis has four main contributions: the adaptation of linear monitors from functional to imperative, the proposal of shared monitors (in an imperative setting) that correspond with the known semantics of linear monitors, thereby extending the notion of partial identity to shared session types, a corresponding implementation of linear and shared monitors in CC0, and case studies which demonstrate the utility and appeal of monitors. We also note that we do not provide operational semantics for the proposed type system of linear and shared monitors and therefore do not prove the usual progress and preservation theorems; we instead rely on informal arguments appealing to intuition.

Section 2 introduces Minimalist CC0, or MCC0, a minimal imperative language with linear and shared session types closely resembling CC0, and its corresponding type system. Section 3 introduces linear monitors and the corresponding additions to the type system. Section 4 introduces shared monitors. Section 5 contains various case studies and examples showcasing both variants of monitors in CC0.

2 Type system and Syntax for MCC0

2.1 Types

We first consider the type system of MCC0.

2.1.1 Ordinary Types

We begin with the ordinary types, τ. The ordinary types are the so-called primitive types. Since we are constructing a minimalist language, the only ordinary type in MCC0 is a **bool**, which is the usual boolean type with two possible values. If one were to extend MCC0 to a more practical language, familiar primitives such as int, float, char, etc. can be included without affecting the typesystem that we later present.

2.1.2 Ordinary Type Constuctors

Although we do not include any non-session type related type constructors for simplicity, we do want to briefly discuss the typical type constructors. Constructors such as structs, arrays, pointers, etc. fall in this category. For the most part, these constructors do not affect our typesystem similar to the addition of more ordinary types like int. That being said, we do want to specify that types that somehow reference memory, such as pointers (and arrays depending on the underlying implementation) would require some care since for example, it does not make sense to send a pointer through a channel; if sent across different machines or encapsulations, the pointer no longer makes sense, and even if sent across processes with shared memory, allowing pointers to be sent would allow an unrestricted form of interprocess communication, undermining the whole point of enforcing a typesystem on interprocess communication. Allowing pointers also further complicates the partial identity check in section 3 where one of the verifications involve checking that certain variables are not mutated.

2.1.3 Linear Session Types

Action As introduced, there are two actions (denoted a); for all sendable types T (formally defined in 2.3), $!T$ means to send something of type T and $?T$ means to receive something of type T. Thus, at its most basic form, a linear session type is a list of actions denoted $a_1; a_2; \cdots; a_n$. However, this is not sufficient; in particular, it is not clear what happens after a_n.

We next introduce a special "action" 1, or the unit, denoting termination of a session. From the provider's point of view, it must close a channel of type 1 and from the user's point of view, it must wait on a channel of type 1 for it to be closed by the provider.

Thus, our second formulation of linear session types consist of a list of non-unit actions (from now on, actions are implicitly assumed to be non-unit) followed by the unit

$$a_1; \cdots; a_n; 1$$

where the implication of this session type is that after performing some action a_n, the provider must terminate, and similarly, the user must wait for termination.

External and Internal Choices Let an arbitrary linear session type be denoted as A. We recall the informally introduced binary operators denoting branching, & and $\oplus$, where $A_1 \mathbin{\&} A_2$ means to offer a choice between A_1 and A_2 and $A_1 \oplus A_2$ means to select a choice between A_1 and A_2. Note that unlike in its earlier introduction, we specifically require these operators to operate on linear session types and not individual actions. These & and $\oplus$ are called external and internal choices respectively since & requires an external decision (user) while $\oplus$ requires the provider to internally make a decision.

We first generalize these operators into their n-ary versions and adopt a new notation for them. To justify this, consider without loss of generality the $\oplus$ operator. A list of linear session types can then all be conglomerated into a series of choices.

$$(A_1 \oplus (A_2 \oplus (\cdots (\oplus A_n) \cdots)))$$

To select A_i, we simply choose the type on the right until we arrive at the type

$$A_i \oplus (A_{i+1} \oplus \cdots))$$

and finally select the left type. Thus these binary operators can be trivially transformed to support arbitrary lengths.

For usability and in the interest of more closely matching the type system of $CC0$, we allow these n-ary choices by the following syntax:

$$!\textbf{choice}\{\overline{(l, A)}\}, ?\textbf{choice}\{\overline{(l, A)}\},$$

where the former denotes an internal choice (since the provider "sends" a choice) whereas the latter denotes an external choice (provider "receives" a choice). The l (label) is a metavariable denoting some identifier. The overline is the usual list notation. Thus, instead of a series of left or right choices like in the binary case, we send and receive a label corresponding to the session type we would like to proceed as.

Since the meaning behind the choices is that after a selection has been made by either the provider ($\oplus$) or the user (&), the session then proceeds as the type that was associated with the selection, there can be no action following a choice. Thus, our third formulation of linear session types is

$$\overline{a}; 1 \mid \overline{a}; !\textbf{choice}\{\cdots\} \mid \overline{a}; ?\textbf{choice}\{\cdots\}$$

Recursive Choices Although finite session types can express many protocols, there are other protocols that are better represented as recursive protocols. For example, recall the random number service that offers a choice of two types and sends a random number of that type. In the type system that we have introduced so far, this can be typed as an external choice (since it receives a choice from the user):

$$?\mathbf{choice}\{(l_1, \tau_1; 1), (l_2, \tau_2; 1)\}$$

for some arbitrary types τ_1, τ_2 (such as **bool**) and labels l_1, l_2.

This protocol implies that the service terminates after sending some value of type τ_1 or τ_2 depending on the choice, but it might make sense for the service to continue for as long as the user requests it. In this case, we would like to implement something of type

$$?\mathbf{choice}\{(l_1, \tau_1; ?\mathbf{choice}\{\cdots\}), (l_2, \tau_2; ?\mathbf{choice}\{\cdots\}, (\mathit{done}, 1))\}$$

where ideally, the inner **choice**$\{\cdots\}$ is the original type. The final *done* choice was added for flavor with the intention that a user might want to request for the service to terminate.

To resolve this, we can allow choices to be named and referenced. Thus, the final type of the protocol can be written as

$$?\mathbf{choice}\ \omega\{(l_1, \tau_1; ?\mathbf{choice}\ \omega), (l_2, \tau_2; ?\mathbf{choice}\ \omega), (\mathit{done}, 1))\}$$

This is reminiscent of the usual recursive type constructor $\mu\rho.\tau$ but restricted for session types. This construction follows CC0, where there were stylistic motivations to allow only choice naming. From a purely functionality standpoint, general recursive types can still be simulated by using a choice with only one option in a recursive manner.

Summary Note that in the following sections, we ignore the subtle details between the three different syntaxes for choice (named choice, reference of a named choice, and anonymous choice) and instead assume an anonymous choice that is type expanded as necessary during typechecking.

Therefore, a linear session type A is denoted as

$$\{\overline{a}; 1\} \mid \{\overline{a}; !\mathbf{choice}\{\overline{(l, A)}\}\} \mid \{\overline{a}; ?\mathbf{choice}\{\overline{(l, A)}\}\}$$

2.1.4 Shared Session Types

Recall that in this paper, we ignore the copy shared session types and only focus on the manifest shared session types.

We assume that these session types are equi-synchronizing, which requires linear channels to be released (upshift) at the same type of the acquire (downshift) if it was originally a shared channel. We defer the formal definition and verification rules to the work by Balzer et al. [BP17], section 3.3 [2]

[2]Since equi-synchronizing is a property of session types and is independent of the underlying structure of the language, the fact that we have deviated from a functional setting where equi-synchronizing was proposed to an imperative setting is irrelevant, justifying this deferral.

In a shared channel, a client must acquire exclusive access of the provider before proceeding and the provider must accept a single acquire request. This is represented with the shift action:
$a ::== \cdots | \#$

Since a shared session type can only downshift to a linear session type and a linear session type can only upshift to a shared session type, instead of introducing two different shift operators, we instead introduce a new notation for shared session types and infer the direction of the shift (up/down) based on whether the session type is linear or shared.

A shared session type S is then of the following form (note the use of square brackets [] instead of curly brackets {}):

$$[\#; \overline{a}; 1] \mid [\#; \overline{a}; !\textbf{choice}\{\overline{(l, A)}\}] \mid [\#; \overline{a}; ?\textbf{choice}\{\overline{(l, A)}\}]$$

Remark Since we defined # to be an action, we can include the initial # as part of $\overline{a}$, but we explicitly showed the initial shift to emphasize that a shared channel can only acquire/accept depending on user/provider.

2.1.5 Persistent Types

So far, we introduced three main categories of types: ordinary, linear session, and shared session. We define the persistent types, κ, to be either ordinary or a shared session. Since linear channels are non-copyable and mutate, it turns out that shared session types are more closely related to ordinary variables in some sense. Therefore, we group the ordinary types and shared session types in the same category for conciseness.

2.1.6 Function and Process Types

A function is something that takes some inputs as arguments and return a value of some ordinary type. On the other hand, a process is something that takes some inputs as arguments and provide a channel of some session type.

Therefore a function returning a value of type τ is of form:

$$(\overline{\kappa}, \overline{A}) \to \tau$$

for persistent types κ and linear session types A.

Similarly, a linear process providing a channel of type A is of form:

$$(\overline{\kappa}, \overline{A}) \to A$$

and a shared process providing a channel of type S is of form:

$$(\overline{\kappa}) \to S$$

It is important to note that a shared process cannot depend on linear channels (as shown by the signature). This is due to the independence principal as explained in [BP17] which only allows linear channels after a downshift.

2.2 Structure

2.2.1 Expression

Expressions are simple mostly due to our intentionally small ordinary types. An expression e is either one of the boolean constants **true**, **false**, a persistent variable x (either an ordinary variable or a shared channel), or an ordinary function call $f(\overline{e})$, which is a specific function call that does not involve linear channels.

Expressions are intentionally intended to be independent of linear channels. Some forms that are typically seen as expressions are instead represented as statements.

2.2.2 Statement

On the other hand, statements are rich. In this section we introduce the interesting statements that mainly deal with linear and shared session types. Other more familiar statements such as variable declaration are listed in the summary table in section 2.3.

Control Flow We begin with control flow related statements. We allow sequential composition of statements $s_1; s_2$ where the intended meaning is that after s_1 executes, s_2 executes. We also allow an if-then-else statement $\mathbf{if}(e)\ \{s_1\}\ \{s_2\}$ and a while loop statement $\mathbf{while}(e)\ \{s\}$ with the usual imperative language semantics. Finally, we include a pattern matching over choices which is later explained in this section along with labels in choices.

Shared Spawn and Aliasing Since shared channels are persistent, we group shared spawn and aliasing with ordinary assignments. In particular, a shared spawn involves invoking a shared channel and assigning its offering channel to an alias:

$$h = s(\overline{e}, \overline{A})$$

where h is a shared channel and s is the name of a shared process.

Shared aliasing on the other hand is a simple assignment of form:

$$h = h'$$

which aliases the channel referenced in h'. Since shared channels allow multiple clients, this aliasing is totally acceptable but is meaningless in terms of functionality since it is comparable to duplicating a constant ordinary variable in that anything that could have been done with h could have been just as easily been done with h'. This is implicitly handled as part of a familiar variable assignment syntax:

$$h = e$$

for some expression e since the only expression that can typecheck with a shared channel h is a shared channel.

Linear Spawn and Rename/Forward We can spawn a new linear process and assign its offering channel to a fresh linear channel in a similar syntax to the shared spawn:

$$d = p(\overline{e}, \overline{A})$$

We can also rename linear channels:

$$d = d'$$

where the distinction here is that since linear channels only allow a single client, d' is no longer available and d simply takes the role of d'.

A special case of the syntactical renaming is a forward:

$$c = d$$

where c is an offering channel in the context of a process and d is some client channel to another process. In this case, this is not a renaming but a signaling that the process will not continue as if it is the process that provides d. Semantically, this statement means that the offering channel c is to be handled by a client channel. A graphical representation of a linear forward is available in figure 2.

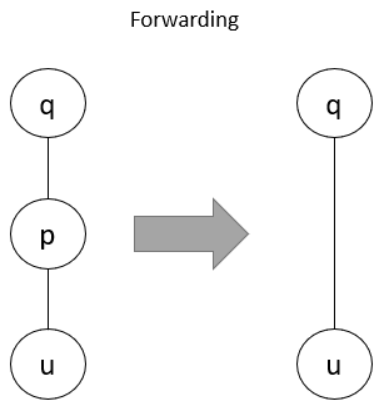

Figure 2: A demonstration of forward by process p. In the context of p, it is offering a channel c which is used by process u and also acting as a client to process q with a channel d. In this context, a forward $c = d$ means that p effectively dissappears; all messages sent by u to p are now forwaded to q and vice versa.

Wait and Close From the client's perspective, if a linear channel is of type $\{1\}$, it must wait until the channel is closed by the provider. This is expressed as **wait**(d) for some channel d. On the other hand, from the provider's perspective, it must close the channel with the statement **close**(c) for its providing channel c.

Send and Receive We previously established the intuitive idea that we either receive or send something for linear channels. We therefore introduce two language primitives for these two operations:

$$\begin{gathered}\mathbf{send}(d, e)\\ \mathbf{send}(d, d')\\ x = \mathbf{recv}(d)\\ d' = \mathbf{recv}(d)\end{gathered}$$

where the first two send expressions (thus persistent types) or a linear channel respectively and the final two similarly receive from a channel and assign to a persistent typed variable x or a fresh linear channel d' respectively. We chose to make $\mathbf{recv}(\cdots)$ not an expression since receiving from a linear channel changes its session type.

Send and Receive Labels Recall that choices consist of a list of pairs (l, A). Thus, the side that is sending a choice (provider if internal choice and user if external choice) must send this choice through a label. A label l can be sent across a channel d with the syntax $d.l$.

To receive a label, we must be ready to account for all possible choices and provide an implementation of each possible case. This is accomplished using a switch-like statement:

$$\mathbf{switch}(d)\ \{\overline{l : s}\}$$

with d as the channel which we receive a label (a choice) from. The switch statement consists of a list of pairs consisting of labels and statements.

Shift Operations We recall that given a shared session type, it can only be downshifted into its corresponding linear session type. Similarly, a linear session type with a shift (#) action can only be upshifted into its corresponding shared session type. Operationally, a downshift requires a client to acquire exclusive access to the shared channel and a provider to accept the request. Similarly, an upshift requires a client to release its exclusive access and the server to then detach itself from the client.

$$\begin{gathered}d = \mathbf{acq}(h)\\ h = \mathbf{rel}(d)\\ c = \mathbf{acc}(g)\\ g = \mathbf{det}(c)\end{gathered}$$

An acquire therefore gives us a fresh linear channel d representing the downshifted channel and release restores the linear channel d back to its shared form. Similarly, an accept gives us a new offering linear channel c representing the shared process's transition to a linear form and a detach restores the linear c to the shared g.

The difference in variable names are reflections of our convention that c is a linear channel a process offers while d is a linear channel that it (or a function) uses and similarly g is a shared offering channel and h is a shared client channel.

2.2.3 Declaration and Program

A declaration *decl* is either a function or a process definition.

A function declaration follows a C-like syntax with additional parameters for contracts:

$$\tau\ f(\overline{\kappa\, x}, \overline{A\, d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}]\{s\}$$

where τ is the return type, f is the function name, $\overline{\kappa\, x}$ are arguments of persistent types (ordinary and shared session), $\overline{A\, d}$ are arguments of linear session types, $\mathbf{req}(e)$ are pre-condition contracts, $\mathbf{ens}(e)$ are post-condition contracts, and finally, s is a statement consisting of the implementation of f.

A process declaration offers a channel and hence has a slightly different form:

$$A\, c\, p(\overline{\kappa\, x}, \overline{A\, d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}]\{s\}$$
$$S\, g\, r(\overline{\kappa\, x})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}]\{s\}$$

where c is the name of the linear offering channel and g is the name of the shared offering channel by our convention. Shared processes cannot receive linear channels by the independence principle.

Finally, a program *prog* consists of a list of declarations *decl*.

2.3 Abstract Syntax Summary

Sort	Abstract Form	Remarks
Metavariable	c, d	Linear channel (offering, using respectively)
	g, h	Shared channel (offering, using respectively)
	x	Persistent variable
	p, q	Linear process name
	r	Shared process name
	f	Function name
	l	Choice names (labels)
	z	Offering channel (either linear or shared)
Program pg	$dcl\, pg$	List of declarations
Decl dcl	$\tau\, f(\overline{\kappa\, x}, \overline{A\, d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}]\{s\}$	Function with contracts
	$A\, c\, p(\overline{\kappa\, x}, \overline{A\, d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}]\{s\}$	Linear process with contracts
	$S\, g\, r(\overline{\kappa\, x})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}]\{s\}$	Shared process with contracts
Stmt s	$\mathbf{return}\ e$	Return ordinary type
	$\tau\ x$	Declare fresh ordinary variable
	$x = e$	Variable assignment
	$x = f(\overline{e}, \overline{d})$	Function call assignment
	$d' = d$	Alias/forward
	$d' = p(\overline{e}, \overline{d'})$	Spawn linear process
	$x = \mathbf{recv}(d)$	Receive ordinary value/shared channel
	$d' = \mathbf{recv}(d)$	Receive linear channel
	$d.l$	Send choice label
	$\mathbf{send}(d, e)$	Send persistent type
	$\mathbf{send}(d, a)$	Send linear channel
	$\mathbf{wait}(d)$	Wait for channel to close
	$\mathbf{close}(c)$	Close channel
	$d = \mathbf{acq}(h)$	Acquire (downshift by client)
	$h = \mathbf{rel}(d)$	Release (upshift by client)
	$d = \mathbf{acc}(h)$	Accept (downshift by provider)
	$h = \mathbf{det}(d)$	Detach (upshift by provider)
	$s_1; s_2$	Sequence of statements
	$\mathbf{if}(e)\ \{s\}\ \{s\}$	If-then-else statement
	$\mathbf{while}(e)\ \{s\}$	While statement
	$\mathbf{switch}(d)\ \{\overline{l : s}\}$	Pattern matching over choices
	$\mathbf{assert}(e)$	Assertion (contract)

Sort	Abstract Form	Remarks
Expression e	$\textbf{true}, \textbf{false}$	Boolean constants
	$f(\overline{e})$	Function call (no channels)
	x	Persistent variable
Persistent type κ	τ	Ordinary Type
	S	Shared session type
Ordinary Type τ	**bool**	Boolean
Signature type *sigtp*	$(\overline{\kappa}, \overline{A}) \to \tau$	Function type
	$(\overline{\kappa}, \overline{A}) \to A$	Linear process type
	$(\overline{\kappa}, \overline{A}) \to S$	Shared process type
Sendable Types T	κ	Persisnt type
	A	Linear session type
Action a	$!T$	Provider sends something of T
	$?T$	Provider receives something of T
	$\#$	Shift point
Linear Session Type A	$\{\overline{a}; 1\}$	Ending with unit
	$\{\overline{a}; !\textbf{choice}\{\overline{(l, A)}\}\}$	Ending with internal choice
	$\{\overline{a}; ?\textbf{choice}\{\overline{(l, A)}\}\}$	Ending with external choice
Shared Session Type S	$\{\#; \overline{a}; 1\}$	Ending with unit
	$\{\#; \overline{a}; !\textbf{choice}\{\overline{(l, A)}\}\}$	Ending with internal choice
	$\{\#; \overline{a}; ?\textbf{choice}\{\overline{(l, A)}\}\}$	Ending with external choice
Session Type Z	A	linear session type
	S	shared session type

2.4 Final Remarks

The minimalist style of this language removes many convenient constructs from CC0 that are effectively sugaring such as for loops (variant of a while loop), if statements without an else clause (an if-then-else with an empty else branch), and simultaneous declaration assignments ($\tau x = e$). Furthermore, ordinary types are extremely limited (only **bool**) for simplicity as mentioned earlier.

Aside from these more obvious differences to CC0, there are some nuanced differences. For example, we are forbidden from referencing linear session variables inside expression level. In particular, $\textbf{recv}(\cdots)$ are now statements in the form of variable assignments and function calls that rely on any linear channels are now lifted to statements in the form of variable assignments. In practical terms, these changes disallow nesting receives and arbitrary functions, which are already undefined behaviors in CC0 due to a fixed evaluation order of function arguments not being fully implemented. We also group shared session types with the ordinary types instead of with linear session types since shared session types more closely resemble ordinary types than linear session types in most cases. We refer readers who are interested in the difference between MCC0 and CC0 to the C0 manual [Pfe18] for further reading [3].

Overall, we justify these simplifications because they would not affect the type judgements of session types, which are the targets of primary interest in this paper.

2.5 Type Judgements

2.5.1 Assumptions

Uniqueness of Identifiers Identifiers, including function and process names, label names, and variable names, are all assumed to be unique in their respective groupings. In particular, function and process names are assumed to be mutually unique, labels are assumed to be unique given a fixed choice (a choice cannot consist of two labels of the same name), persistent variables (ordinary variables and shared channels) names must be mutually unique, and linear channels must be unique with respect to itself. As an important note, linear channel name being unique also means that the offering channel name and client channel names must also be mutually unique.

Global Function/Process table We assume an initial pass over the program to construct a global list Σ consisting of all function and process signatures. This means that all functions and processes are accesible from anywhere and forward declarations of signatures are not only unavailable but also not required.

2.5.2 Expressions

As a result of an imperative style language with its primary focus on linear session types, which are typed inside statement level, expression checking is concise. However, we do note that adding practical programming language constructs that were intentionally omitted such as operators would mainly populate expression judgement rules, which of course will not interact with any of the session type checking done on statement level.

We first introduce the usual context consisting of the persistent types:

$$\Gamma \ ::= \ \cdot \mid \Gamma, x : \kappa$$

An important point is that we do not have any context for the linear channels when typechecking expressions. This is the due to our stylistic motivation that expressions should be independent of session types and closely resemble expressions in typical imperative languages without session types. The fact that we allow shared channels inside expressions is fine because shared channels operate similar to ordinary variables.

[3]Note that a CC0 reference manual is not available, so this manual will only cover the non-session type related distinctions between MCC0 and CC0

In general, an expression judgement is of form

$$\Gamma \vdash e :: \kappa$$

where the judgement reads: in a context Γ, expression e is judged to be of type κ.

We first introduce rules for **true**, **false**, and variables.

$$\frac{}{\Gamma \vdash \textbf{true} :: \textbf{bool}} \text{(EXP-TRUE)}$$

$$\frac{}{\Gamma \vdash \textbf{false} :: \textbf{bool}} \text{(EXP-FALSE)}$$

$$\frac{}{\Gamma, x : \kappa \vdash x :: \kappa} \text{(EXP-VAR)}$$

Finally, function calls without linear channels requires the signature of f as found in the global Σ to match with its usage.

$$\frac{\Gamma \vdash \overline{e} :: \overline{\kappa} \qquad f : (\overline{\kappa}) \to \tau \in \Sigma}{\Gamma \vdash f(\overline{e}) :: \tau} \text{(EXP-FCALL)}$$

2.5.3 Statements

To typecheck statements, we maintain the same persistent context Γ as in the expression judgements:

$$\Gamma ::= \cdot \mid \Gamma, x : \kappa$$

and a linear context Δ:

$$\Delta ::= \cdot \mid \Delta, d : A$$

Linear Context Structurally, Δ and Γ are exactly the same except for the underlying types; Γ stores the typing of persistent variables while Δ similarly stores the typing of linear channels (linear session type variables). However, they operate very differently because of the distinction that the types of persistent variables are permanent whereas the types of linear channels are ephemeral.

Consider for example some client channel $d : \{?\textbf{bool}; a; \cdots\}$. Since it is a client channel, the first action $?\textbf{bool}$ requires us to send a value of type **bool**. After executing a statement that sends across d, for example

$$\textbf{send}(d, \textbf{true})$$

d "progresses" – d is now of type $\{a; \cdots\}$.

This property of linear channels differentiate it from the persistent variables which cannot change in type. Thus, we require a two-state formulation as used by Hodas et al. [HPC96]:

$$\Delta \setminus \Delta'$$

where the intended meaning in the later introduced statement judgement rule is that Δ is the linear context before execution of some statement and Δ' is a potentially different linear context after execution of the statement. Γ, on the other hand, does not require a two-state formulation and therefore operates in the usual way.

Statements need to be aware of the context of its declaration. For example, to typecheck **return** e, we need to know that this statement is part of a function declaration and its return type τ. Similarly for processes, we need to know the offering channel name z and its type Z. Although the return type of a function does not change, the analogous offering linear session type of a linear process does change with the same reasoning as for the linear context. As for shared processes, since an accept somehow "transforms" itself into an exclusive access mode where it acts linearly and changes its name, both the session type and the name of the offering channel changes. Therefore, we require a two-state formulation of the offering channel as well:

$$((z : Z \ \setminus \ z' : Z'))$$

Next, we define a statement judgement to be in a terminated state if the end of control flow is reached, which means that operationally, no statements can further be executed. This distinction becomes necessary when we typecheck control flow statements like the if-then-else statement; if one branch succesfully enters a terminated state while the other does not, typechecking on subsequent statements should proceed from the state that the non-terminating branch ended on. We allow a statement causing termination to be expressed as part of the linear context and the offering channel (only linear context if function since it has no offering channel):

$$\Delta \setminus \downarrow$$
$$(z : Z \ \setminus \ z' : \uparrow)$$

We next introduce two meta-variables $\Delta_\downarrow$ and $Z_\uparrow$:

$$\begin{aligned} \Delta_\downarrow &::= \Delta \mid \downarrow \\ Z_\uparrow &::= Z \mid \uparrow \end{aligned}$$

where $\downarrow$ captures a terminated state through the linear context and $\uparrow$ captures a terminated state through the offering channel.

A statement judgement therefore is of two possible forms: function and process:

$$\begin{aligned} &\Gamma \ ; \ \Delta \ \setminus \ \Delta'_\downarrow \ \vdash s :: \tau \\ &\Gamma \ ; \ \Delta \ \setminus \ \Delta'_\downarrow \ \vdash s :: (z : Z \ \setminus \ z' : Z'_\uparrow) \end{aligned}$$

Because many judgement rules do not necessarily need to use the right side of the double colon, we introduce a meta-variable Θ:

$$\Theta \ ::= \ \tau \mid (z : Z \ \setminus \ z' : Z'_\uparrow)$$

which can replace the two contexts of declaration. Therefore, a general statement judgement is of form:

$$\Gamma \ ; \ \Delta \ \setminus \ \Delta'_\downarrow \ \vdash s :: \Theta$$

We first introduce the two subsumption rules. As described when introducing termination rules, this is used in control flow operations; when one side of a branch (for example, due to an if statement) terminates, we allow the side of termination to construct an arbitrary state such that we can continue from the state of the other branch.

$$\frac{\Gamma\ ;\ \Delta\ \setminus\ \downarrow\ \vdash s :: \tau}{\Gamma\ ;\ \Delta\ \setminus\ \Delta'\ \vdash s :: \tau}\ (\text{STM-TERMA})$$

$$\frac{\Gamma\ ;\ \Delta\ \setminus\ \downarrow\ \vdash s :: (c : A\ \setminus\ c : \uparrow)}{\Gamma\ ;\ \Delta\ \setminus\ \Delta'\ \vdash s :: (c : A\ \setminus\ c : A')}\ (\text{STM-TERMB})$$

Next we deal with assignment to persistent variables. This includes assignments on ordinary variables and also shared channels. Therefore, the two rules also account for aliasing and spawning on shared channels respectively. Note that upshifts are later expressed as special cases. Note that in (STM-ASG-F), linear channels that are transferred to the function f are no longer available in our context. This makes sense since linear channels only allow one client.

$$\frac{\Gamma, x : \kappa\ \vdash e :: \kappa}{\Gamma, x : \kappa\ ;\ \Delta\ \setminus\ \Delta\ \vdash x = e :: \Theta}\ (\text{STM-ASG-E})$$

$$\frac{\Gamma, x : \tau\ \vdash \overline{e} :: \overline{\kappa} \qquad f : (\overline{\kappa}, \overline{A}) \to \tau \in \Sigma}{\Gamma, x : \tau\ ;\ \Delta, \overline{d : A}\ \setminus\ \Delta\ \vdash x = f(\overline{e}, \overline{d}) :: \Theta}\ (\text{STM-ASG-F})$$

A return can only occur inside a function and the type must match the return type of the function it is in.

$$\frac{\Gamma\ \vdash e :: \tau}{\Gamma\ ;\ \cdot\ \setminus\ \downarrow\ \vdash \textbf{return}\ e :: \tau}\ (\text{STM-RET})$$

Since our language cannot have any side effects from a boolean expression due to expressions not being able to operate on channels, we do not need to check for any notion of "effectless" inside assertions. Thus, we only need to check that the inner expression is a predicate; that is, a **bool**. A more productive language would need to further check that e is pure; it contains no side effects.

$$\frac{\Gamma\ \vdash e :: \textbf{bool}}{\Gamma\ ;\ \Delta\ \setminus\ \Delta\ \vdash \textbf{assert}(e) :: \Theta}\ (\text{STM-ASSERT})$$

We finally move on to session types. We begin with the simplest rules. A process offering some linear channel c is only allowed to close itself if it is not a client to other linear processes, meaning the linear context Δ needs to be empty. Closing also puts itself in a termination state, represented by the up and down arrows in the respective locations.

$$\frac{}{\Gamma\ ;\ \cdot\ \setminus\ \downarrow\ \vdash \textbf{close}(c) :: (c : \{1\}\ \setminus\ c : \uparrow)}\ (\text{STM-CLOSE})$$

On the other hand, a wait simply causes the channel d to disappear from the context after execution. The Θ on the right hand side means that this rule applies to both function and process contexts.

$$\frac{}{\Gamma\ ;\ \Delta, d : \{1\}\ \setminus\ \Delta\ \vdash \textbf{wait}(d) :: \Theta}\ (\text{STM-WAIT})$$

Next, we introduce the send rules for the providing channel from a linear process. Since we view session types from a provider perspective, an action of form $!T$ on the offering channel means that we must send a value of that type. Note that when we send a channel (STM-SEND-CH1), the channel that we send is no longer available in our linear context – this is not surprising because sending a linear channel is effectively transfering the connection since we can only have one client for a given linear channel.

$$\frac{\Gamma \vdash e :: \kappa}{\Gamma\ ;\ \Delta \ \backslash\ \Delta \vdash \mathbf{send}(c, e) :: (c : \{!\kappa; \overline{a}\} \ \backslash\ c : \{\overline{a}\})} \text{(STM-SEND-PERS1)}$$

$$\frac{}{\Gamma\ ;\ \Delta, d : A \ \backslash\ \Delta \vdash \mathbf{send}(c, d) :: (c : \{!A; \overline{a}\} \ \backslash\ c : \{\overline{a}\})} \text{(STM-SEND-CH1)}$$

On the other hand, an offering channel with an action of form $?T$ requires us to receive something of type T. Analogous to the above send rules, we have one rule each for the persistent types and linear session types.

$$\frac{}{\Gamma, x : \kappa\ ;\ \Delta \ \backslash\ \Delta \vdash x = \mathbf{recv}(c) :: (c : \{?\kappa; \overline{a}\} \ \backslash\ c : \{\overline{a}\})} \text{(STM-RECV-PERS1)}$$

$$\frac{}{\Gamma\ ;\ \Delta \ \backslash\ \Delta, d : A \vdash d = \mathbf{recv}(c) :: (c : \{?A; \overline{a}\} \ \backslash\ c : \{\overline{a}\})} \text{(STM-RECV-CH1)}$$

Sending a label through the offering channel requires the session type to be an internal choice. After selection, the offering channel continues as the linear session type corresponding to the label that it sent.

$$\frac{}{\Gamma\ ;\ \Delta \ \backslash\ \Delta \vdash c.l_i :: (c : \{!\mathbf{choice}\{\overline{l, A}\}\} \ \backslash\ c : A_i)} \text{(STM-LABEL1)}$$

On the other hand, we can receive a label through the offering channel if the session type is an external choice. This introduces a pattern matching over all possible labels in the external choice. An important note is that the offering channel c begins as A_i in all the individual cases by the semantics of an external choice. The premise also requires that all branches end up with the same linear context and offering channel type; the subsumption rule will internally be used to satisfy this premise as necessary if any of the individual cases enter a terminated state. Note that the metavariables $\Delta'_{\downarrow}$ and $A'_{\uparrow}$ are still being used since all branches can lead to termination, in which case we would like to propagate the information that the termination state has been reached.

$$\frac{\forall i\ (\Gamma\ ;\ \Delta \ \backslash\ \Delta'_{\downarrow} \vdash s_i :: (c : A_i \ \backslash\ c : A'_{\uparrow}))}{\Gamma\ ;\ \Delta \ \backslash\ \Delta'_{\downarrow} \vdash \mathbf{switch}(c)\ \{\overline{l : s}\} :: (c : \{?\mathbf{choice}\{\overline{l, A}\}\} \ \backslash\ c : A'_{\uparrow})} \text{(STM-SWITCH1)}$$

The following rules dualize the previous rules on the offering channel to an arbitrary client channel inside our linear context. Recall that since we view session types from a provider perspective, the meaning of ! and ? are flipped from the provider's meaning when typechecking client channels.

$$\frac{\Gamma \vdash e :: \kappa}{\Gamma\ ;\ \Delta, d : \{?\kappa; \overline{a}\} \ \backslash\ \Delta, d : \{\overline{a}\} \vdash \mathbf{send}(d, e) :: \Theta} \text{(STM-SEND-PERS2)}$$

$$\frac{}{\Gamma\ ;\ \Delta, d : \{?A; \overline{a}\}, d' : A \ \backslash\ \Delta, d : \{\overline{a}\} \vdash \mathbf{send}(d, d') :: \Theta} \text{(STM-SEND-CH2)}$$

$$\frac{}{\Gamma, x : \kappa\ ;\ \Delta, d : \{!\kappa; \overline{a}\} \ \backslash\ \Delta, d : \{\overline{a}\} \vdash x = \mathbf{recv}(d) :: \Theta} \text{(STM-RECV-PERS2)}$$

$$\frac{}{\Gamma\ ;\ \Delta, d : \{!A; \overline{a}\}\ \setminus\ \Delta, d : \overline{a}, d' : A\ \vdash d' = \mathbf{recv}(d) :: \Theta}\ \text{(STM-RECV-CH2)}$$

$$\frac{}{\Gamma\ ;\ \Delta, d : \{?\mathbf{choice}\{\overline{l, A}\}\}\ \setminus\ \Delta, d : A_i\ \vdash d.l_i :: \Theta}\ \text{(STM-LABEL2)}$$

$$\frac{\forall i\ (\Gamma\ ;\ \Delta, d : A_i\ \setminus\ \Delta'_{\downarrow}\ \vdash s_i :: \tau)}{\Gamma\ ;\ \Delta, d : \{!\mathbf{choice}\{\overline{l, A}\}\}\ \setminus\ \Delta'_{\downarrow}\ \vdash \mathbf{switch}(d)\ \{\overline{l : s}\} :: \tau}\ \text{(STM-SWITCH2A)}$$

$$\frac{\forall i\ (\Gamma\ ;\ \Delta, d : A_i\ \setminus\ \Delta'_{\downarrow}\ \vdash s_i :: (z : Z\ \setminus\ z' : Z'_{\uparrow}))}{\Gamma\ ;\ \Delta, d : \{!\mathbf{choice}\{\overline{l, A}\}\}\ \setminus\ \Delta'_{\downarrow}\ \vdash \mathbf{switch}(d)\ \{\overline{l : s}\} :: (z : Z\ \setminus\ z' : Z'_{\uparrow})}\ \text{(STM-SWITCH2B)}$$

In an imperative language, variable declarations are handled as special cases to statement sequences. We also highlight that whereas shared channels, being a persistent variable, need to be declared, linear channels do not.

$$\frac{\Gamma, x : \kappa\ ;\ \Delta\ \setminus\ \Delta_{\downarrow}\ \vdash s :: \Theta}{\Gamma\ ;\ \Delta\ \setminus\ \Delta_{\downarrow}\ \vdash \kappa\, x; s :: \Theta}\ \text{(STM-VDECL)}$$

Sequences are typechecked component-wise. It is important to note that when checking $s_1; s_2$, we must check s_2 with the state after executing s_1.

$$\frac{\Gamma\ ;\ \Delta_1\ \setminus\ \Delta_2\ \vdash s_1 :: \tau \qquad \Gamma\ ;\ \Delta_2\ \setminus\ \Delta_{\downarrow}\ \vdash s_2 :: \tau}{\Gamma\ ;\ \Delta_1\ \setminus\ \Delta_{\downarrow}\ \vdash s_1; s_2 :: \tau}\ \text{(STM-SEQA)}$$

$$\frac{\Gamma\ ;\ \Delta_1\ \setminus\ \Delta_2\ \vdash s_1 :: (z : Z_1\ \setminus\ z' : Z_2) \qquad \Gamma\ ;\ \Delta_2\ \setminus\ \Delta_{\downarrow}\ \vdash s_2 :: (z' : Z_2\ \setminus\ z'' : Z_{\uparrow})}{\Gamma\ ;\ \Delta_1\ \setminus\ \Delta_{\downarrow}\ \vdash s_1; s_2 :: (z : Z_1\ \setminus\ z' : Z_{\uparrow})}\ \text{(STM-SEQB)}$$

If-then-else statements are actually simpler forms of the pattern matching switch. We require that both branches lead to the same state. Again, if one side terminates but the other does not, the subsumption rules allow us to continue as the side that does not terminate. Similar to the switch rules, we propagate termination state if both branches terminate, which explains the use of the metavariables $\Delta'_{\downarrow}$ and $Z'_{\uparrow}$.

$$\frac{\Gamma\ \vdash e :: \mathbf{bool} \qquad \Gamma\ ;\ \Delta\ \setminus\ \Delta'_{\downarrow}\ \vdash s_1 :: \tau \qquad \Gamma\ ;\ \Delta\ \setminus\ \Delta'_{\downarrow}\ \vdash s_2 :: \tau}{\Gamma\ ;\ \Delta\ \setminus\ \Delta'_{\downarrow}\ \vdash \mathbf{if}(e)\ \{s_1\}\ \{s_2\} :: \tau}\ \text{(STM-IFA)}$$

$$\frac{\Gamma\ \vdash e :: \mathbf{bool} \qquad \Gamma\ ;\ \Delta\ \setminus\ \Delta'_{\downarrow}\ \vdash s_1 :: (z : Z\ \setminus\ z' : Z'_{\uparrow}) \qquad \Gamma\ ;\ \Delta\ \setminus\ \Delta'_{\downarrow}\ \vdash s_2 :: (z : Z\ \setminus\ z' : Z'_{\uparrow})}{\Gamma\ ;\ \Delta\ \setminus\ \Delta'_{\downarrow}\ \vdash \mathbf{if}(e)\ \{s_1\}\ \{s_2\} :: (z : Z\ \setminus\ z' : Z'_{\uparrow})}\ \text{(STM-IFB)}$$

To typecheck a while statement, we first highlight the fact that the loop can execute all the way from zero times to arbitrary many times. Since we cannot statically infer this in the general sense, we require that a single execution of the loop to maintain the state. Thus, linear channels are in some sense loop invariant. Of course, a single execution of the loop can instead terminate the process – this is again allowed thanks to the subsumption rule. Unlike the other control flow statements, we do not propagate any termination information from the inner statement since the loop can still not execute a single time, meaning we must end up at the same state from the while statement.

Since a loop iteration does not allow linear session types to progress, it may seem natural to question its utility in the context of session types. The only way linear session types can be meaningfully used in while loops is when the underlying session type is recursive; in this case, it is possible for a loop iteration to return whatever channels it uses to the original type.

$$\frac{\Gamma \vdash e :: \textbf{bool} \qquad \Gamma\ ;\ \Delta \setminus \Delta_{\downarrow} \vdash s :: \tau}{\Gamma\ ;\ \Delta \setminus \Delta \vdash \textbf{while}(e)\ \{s\} :: \tau}\ \text{(STM-WHILEA)}$$

$$\frac{\Gamma \vdash e :: \textbf{bool} \qquad \Gamma\ ;\ \Delta \setminus \Delta \vdash s :: (z : Z \setminus z : Z)}{\Gamma\ ;\ \Delta \setminus \Delta \vdash \textbf{while}(e)\ \{s\} :: (z : Z \setminus z : Z)}\ \text{(STM-WHILEB)}$$

Since a forwarding effectively ends further execution of the process, it is considered to enter a termination state. To forward, we must only have one linear channel as client similar to the condition imposed in (STM-CLOSE).

$$\frac{}{\Gamma\ ;\ d : A \setminus \downarrow\ \vdash c = d :: (c : A \setminus c : \uparrow)}\ \text{(STM-FWD)}$$

A tailcall $c = p(\cdots)$ is effectively a sugar for $d = p(\cdots); c = d$. Hence, we require that we transfer all client channels to the process we continue as to mimic the requirement imposed in (STM-FWD).

$$\frac{\Gamma \vdash \overline{e} :: \overline{\kappa} \qquad p : (\overline{\kappa}, \overline{A}) \to A' \in \Sigma}{\Gamma\ ;\ \overline{d : A} \setminus \downarrow\ \vdash c = p(\overline{e}, \overline{d}) :: (c : A' \setminus c : \uparrow)}\ \text{(STM-TAILCALL)}$$

Renaming a channel simply replaces itself with its new name in the linear context.

$$\frac{}{\Gamma\ ;\ \Delta, d' : A \setminus \Delta, d : A \vdash d = d' :: \Theta}\ \text{(STM-RENAME)}$$

Spawning a process grants a fresh channel. The typechecking is mostly similar to function calls. Note that linear channels that is transferred to the spawning process are no longer available in our context, similar to (STM-ASG-F). Unlike with shared channels, we do not need to forward declare linear channels and therefore, they simply appear in the linear context whenever spawned. Furthermore, since linear channels are not allowed to depend on shared channels, a process offering shared channel cannot spawn (with the implication that once it "turns linear" after accepting, this restriction is removed).

$$\frac{\Gamma \vdash \overline{e} :: \overline{\kappa} \qquad p : (\overline{\kappa}, \overline{A}) \to A' \in \Sigma}{\Gamma\ ;\ \Delta, \overline{d : A} \setminus \Delta, d' : A' \vdash d' = p(\overline{e}, \overline{d}) :: \tau}\ \text{(STM-SPAWN1)}$$

$$\frac{\Gamma \vdash \overline{e} :: \overline{\kappa} \qquad p : (\overline{\kappa}, \overline{A}) \to A' \in \Sigma}{\Gamma\ ;\ \Delta, \overline{d : A} \setminus \Delta, d' : A' \vdash d' = p(\overline{e}, \overline{d}) :: (c : A_c \setminus c : A_c)}\ \text{(STM-SPAWN2)}$$

Below we have the two shift statements for client shared channels. An acquire creates a fresh linear channel of corresponding type. And a release removes the linear channel from the context. Since acquire creates a fresh channel, we must be in a non-shared context.

$$\frac{}{\Gamma, h : [\#; \overline{a}]\ ;\ \Delta \setminus \Delta, d : \{\overline{a}\} \vdash d = \textbf{acq}(h) :: \tau}\ \text{(STM-ACQ1)}$$

$$\frac{}{\Gamma, h : [\#; \overline{a}]\ ;\ \Delta \setminus \Delta, d : \{\overline{a}\} \vdash d = \textbf{acq}(h) :: linoff A_c A_c}\ \text{(STM-ACQ2)}$$

$$\frac{}{\Gamma, h : [\#; \overline{a}]\ ;\ \Delta, d : \{\#; \overline{a}\} \setminus \Delta \vdash h = \textbf{rel}(d) :: \Theta}\ \text{(STM-REL)}$$

For offering shared channels, an accept changes the offering channel to the linear version. Thus, typechecking then proceeds as if a linear process until it is restored through a detach, where at this point, there must be no linear channels in context.

$$\frac{}{\Gamma\ ;\ \cdot\ \setminus\ \cdot\ \vdash c = \mathbf{acc}(g) :: (g : [\#;\overline{a}]\ \setminus\ c : \{\overline{a}\})}\ \text{(STM-ACC)}$$

$$\frac{}{\Gamma\ ;\ \cdot\ \setminus\ \cdot\ \vdash g = \mathbf{det}(c) :: (c : \{\#;\overline{a}\}\ \setminus\ g : [\#;\overline{a}])}\ \text{(STM-DET)}$$

2.5.4 Declarations

Although function and process contracts appear as special attributes in the definition, they can instead be thought of as syntactic sugar for statement level asserts [4]. Since **req**(e) is a pre-condition, it is checked as an **assert**(e). For **ens**(e), we similarly check it as an **assert**(e) but also include a fixed variable *result* representing the return type of the function if appropriate. To save space, we will introduce the declaration rules without any declaration-level contracts. We describe a pass to convert all declarations with contracts into an equivalent set of declarations without contracts in section 3.3.4 which justifies this decision.

$$\frac{\overline{x:\kappa}\ ;\ \overline{d:A}\ \setminus \downarrow\ \vdash s :: \tau}{\tau\ f(\overline{\kappa\, x}, \overline{A\, d})\{s\}\ \mathbf{decl}}\ \text{(DECL-FUNC)}$$

$$\frac{\overline{x:\kappa}\ ;\ \overline{d:A}\ \setminus \downarrow\ \vdash s :: (c : A'\ \setminus\ c : \uparrow)}{A'\ c\, p(\overline{\kappa\, x}, \overline{A\, d})\{s\}\ \mathbf{decl}}\ \text{(DECL-LPROC)}$$

$$\frac{\overline{x:\kappa}\ ;\ \overline{d:A}\ \setminus \downarrow\ \vdash s :: (g : S\ \setminus\ g' : S)}{S\, g\, r(\overline{\kappa\, x})\{s\}\ \mathbf{decl}}\ \text{(DECL-SPROC1)}$$

$$\frac{\overline{x:\kappa}\ ;\ \overline{d:A}\ \setminus \downarrow\ \vdash s :: (g : S\ \setminus\ g' : \uparrow)}{S\, g\, r(\overline{\kappa\, x})\{s\}\ \mathbf{decl}}\ \text{(DECL-SPROC2)}$$

where statement judgements are introduced in section 2.5.3 and expression judgements are introduced in section 2.5.2.

2.5.5 Program

Since we assume a global table, we do not require any context at program level.

$$\frac{}{\cdot\ \mathbf{prog}}\ \text{(PROG-EMPTY)}$$

$$\frac{dcl\ \mathbf{decl} \qquad prg\ \mathbf{prog}}{dcl\ prg\ \mathbf{prog}}\ \text{(PROG-DCL)}$$

where declaration judgements are introduced in section 2.5.4.

[4]This is not totally true since function and process contracts can be specifically used to obtain blame assignment; a pre-condition violation can give an error message pointing to the malformed caller and so on. We ignore this distinction in this thesis since we do not deal with blame assignment on monitors.

2.6 Example

2.6.1 Remark

Although the formal rules will be written in terms of MCC0, many non-trivial examples of code will be introduced in CC0 for aesthetic reasons. Here, we point out some stylistic and syntactic differences between MCC0 and CC0. In general, as per keeping the specification of MCC0 as simple as possible, many "shortcuts" that are desirable for programmers are not available, so examples in CC0 would be cleaner and more understandable.

In CC0	In MCC0	Notes
$\$d, \$c$	d, c	Identifiers of linear channels in CC0 are preceded by a dollar sign
$\#g, \#h$	g, h	Identifiers of shared channels in CC0 are preceded by a hash
$d = p(d, \dots)$	$d' = p(d, \dots); d = d'$	CC0 can infer that d is no longer available on the right hand side and thus allows an immediate assignment
$\kappa\, x = \dots$	$\kappa\, x; x = \dots$	CC0 allows an assignment on declaration
typedef $T\ n$	N/A	CC0 allows a familiar typedef to bind types to names during compilation
$< \dots ; >$	$\{\dots ; 1\}$	The final unit in session types is inferred in CC0.
choice n	**choice**$\{\cdots\}$	In CC0, choices cannot be defined inline and instead are defined at declaration level. Thus, choices in CC0 reference defined choice names instead of contructing it inline.

2.6.2 Parallel Fibonacci

Listing 2: Parallel Fibonacci in CC0 with typechecking annotation.

```
<!int;> $c fib(int n)
{
    if (n == 0 || n == 1) {
        send($c, 1); // ... |- send($c, 1) :: ($c : <!int;> \ $c : <>)
        close($c);   // ... |- close($c) :: ($c : <> \ $c : up)
    } else {
        <!int;> $f1 = fib(n-1); // ...; $f1 : <!int;> |- s :: (theta)
        <!int;> $f2 = fib(n-2); // ...; $f1 : <!int;>, $f2 : <!int;> |- ...
        int x1 = recv($f1);     // ...; $f1 : <>, $f2 : <!int;> |- ...
        wait($f1);              // ...; $f2 : <!int;> |- ...
        int x2 = recv($f2);     // ...; $f2 : <> |- ...
        wait($f2);              // ...; . |- ... (Delta empty)
        send($c, x1 + x2); // ... |- send($c, x1 + x2) :: ($c : <!int;> \ $c : <>)
        close($c); // ... |- close($c) :: ($c : <> \ $c. up)
15     }
16 }
```

We will briefly walk through the typechecker for this relatively simple linear process. Although the annotation gives reasonable detail in the inner state of the typechecker, the two state formation for Δ was not used due to space reasons. Therefore, the annotations in lines 7 to 12 should be interpreted as the result of Δ after the execution of the corresponding statement, or in other words, the right side of the two state formation.

At line 2, Γ solely consists of n of type int. Δ is empty because there are no linear channels in the argument. The process offers a channel of type $\{!int; 1\}$ (as a reminder, the final unit is inferred in CC0), meaning we expect to send an int and then terminate.

At line 3, we enter an if statement meaning that we now check that both cases terminate with the same state. At line 4 and 5, we send an int and then terminate. The termination is valid because $\$c$ is of type $\{1\}$ and Δ is empty. At this point, the linear context is $\downarrow$ and the offering is $\uparrow$.

At line 7 and 8, we locally spawn two processes and bind them to the linear channels $\$f1$ and $\$f2$. Therefore,

$$\Delta = \$f1 : \{!int; 1\}, \$f2 : \{!int; 1\}$$

At line 9, we bind a receive result from $\$f1$ to $x1$ and wait for termination of $\$f1$.

By line 13, both local processes have succesfully terminated, and Δ is again empty. Γ on the other hand consists of $n, x1, x2$ all of type int. Finally on line 13 and 14, we send and close $\$c$ like on line 4 and 5. The termination is valid because Δ is empty.

Since both branches terminated, the if statement is also valid since the two states are equivalent. The if statement itself is then considered to have terminated since both sides terminate, meaning the process is succesfully typechecked.

Note that if we remove termination from one branch, for instance the if branch on line 5, then the subsumption rule applied to the else branch (since it terminated) would allow us to continue the typechecking as if we are continuing on the if branch, meaning the typechecker would correctly require that after the if else statement, $\$c$ is closed.

3 Linear Monitors and Partial Identity

3.1 Introduction

So far, we have introduced ordinary contracts as statements that, aside from on contract violation, have no observable effect to the outcome of the program.

For ordinary variables, we can "look into" the variable to obtain its value and therefore have the ability to write productive contracts as boolean expressions. However for channels, there is no notion of "value", and so it does not seem possible to express any meaningful contracts as mere boolean expressions.

As we noted previously, Gommerstadt et al. formulate linear session typed contracts as intermediary processes (monitor) that attach to a channel and listen along the channel [GJP18]. See diagram 3 for a visual representation of the monitor.

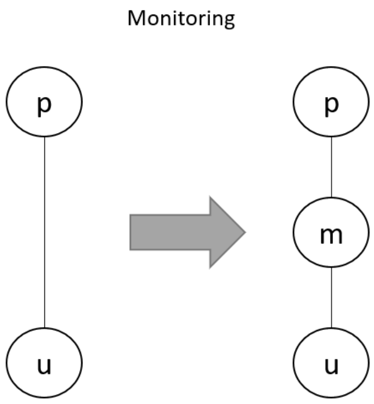

Figure 3: A demonstration of a linear monitor m acting as an eavesdropper between a user u and its client channel provided by p.

Like contracts, monitors cannot have any observable effect on the program, meaning they must faithfully pass all messages they receive, unless a contract is violated. When a process meets this requirement, it is said to have the property of partial identity. Thus, a valid monitor can be formally stated as a partial identity process and session type contracts are simply statements that attach a monitor to an existing channel.

3.2 Language

We will now augment the language with monitors over linear channels:

$$\begin{aligned}
decl \ &::= \tau\, f(\overline{\kappa\, x}, \overline{A\, d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\} \\
&\quad \mid A\, c\, p(\overline{\kappa\, x}, \overline{A\, d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\} \\
&\quad \mid S\, g\, r(\overline{\kappa\, x})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\} \\
&\quad \mid \mathbf{mon}(A\ c\ p(A\, d, \overline{\tau\, x})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\}) \\
s \ &::= \ \ldots \\
&\quad \mid \mathbf{mon}(d = p(d, \overline{e})) \\
sigtp \ &::= \ \ldots \\
&\quad \mid \mathbf{mon}((A, \overline{tp}) \to A)
\end{aligned}$$

All declarations now support monitors over their parameters and providing channel if appropriate. Note that unlike ordinary contracts where there is a syntactic distinction between pre-condition (requires) and post-condition (ensures) contracts, there is no syntactic distinction between monitors that correspond to requires and monitors corresponding to ensures. This is discussed in section 3.3.4. The final definition above allows construction of a linear monitor, which is a subset of linear processes. Note that there is only one linear channel as input; the rest are ordinary values (shared channels are not allowed). The addition of shared channels as parameters in monitors require a more sophisticated typesystem which we briefly discuss in section 6. In practice, the explicit annotation of monitors can instead be inferred with a single pass over the program to detect any processes used as monitors.

Statements now support monitors similar to an assert. The syntax and usage is similar to those in declarations. A compiler should simply transform $\mathbf{mon}(d = m(d, \ldots))$ to $d' = m(d, \ldots); d = d'$ in a debug-build with contracts enabled which effectively inserts the monitor p as in figure 3.

Finally, a valid linear monitor's typing signature will explicitly be marked as such for simplification. In practice, a global table with a listing of all valid linear monitors will serve the same purpose. Although in MCC0 monitors are treated differently from processes, in CC0 they are instead treated as a subset of processes. In particular, this means we can use monitors outside contracts to simulate tests that "always run."

3.3 Type Judgements

3.3.1 Statements

First we amend a rule for statement monitors. Since we explicit mark monitors and check them separately, we simply assume monitors that are marked as such are valid to use; either the monitor has already been checked or it will be checked in the future.

$$\frac{\Gamma \vdash \overline{e} :: \overline{\tau} \qquad p : \mathbf{mon}((A, \overline{\tau}) \to A) \in \Sigma}{\Gamma \ ; \ \Delta, d : A \ \setminus \ \Delta, d : A \vdash \mathbf{mon}(d = p(d, \overline{e})) :: \Theta} \text{(STM-MONITOR)}$$

3.3.2 Partial Identity

Recall that we introduced partial identity previously as a property of linear processes; a linear process with one providing channel and one client channel that satisfies partial identity is one that faithfully forwards all messages between the providing and client channels except on contract violation. Thus, partial identity check is a form of typechecking with much stronger conditions that only runs on monitors.

Intuitively, partial identity can be checked by maintaining a queue of messages to be sent by either the providing channel c or the client d depending on the current direction of the messages and requiring the process to adhere to it. For example, if we receive two values consecutively from the client, then we remember that we need to send the two values that we received to the provider and put it in a queue. Thus, we need to consider what types of messages we can receieve in general; we can either receieve persistent variables, linear channels, labels (for choices), or termination (only from client channel), so we define a member of the queue as follows:

$$q \ ::= \ \mathbf{Pers}(x) \mid \mathbf{Ch}(d) \mid \mathbf{Label}(l) \mid \mathbf{End}$$

Next, we introduce a tracking context which keeps track of the providing and the client end along with the current message direction:

$$tc \ ::= \ (c : A_{\uparrow} \Leftarrow d : A'_{\downarrow}) \mid (c : A_{\uparrow} \Rightarrow d : A'_{\downarrow})$$

where $(\cdots \Leftarrow \cdots)$ means that the message is flowing from d to c; we are currently receiving from the client d and the queue of messages are messages we need to send to c. On the other hand, $(\ldots \Rightarrow \ldots)$ means that the message is flowing from c to d, meaning we are currently receiving from c and therefore our queue consist of messages we need to send to d.

We also introduce two metavariables:

$$\Leftrightarrow \ ::= \ \Leftarrow \ \mid \ \Rightarrow$$
$$\omega \ ::= \ (c : A \Leftrightarrow d : A')$$

where $\Leftrightarrow$ represents some direction (either left or right) and ω represents some tracking context.

Since both the queue and tracking context can change from a statement, they will be in a two state formation as with the linear context Δ.

In summary, a partial identity judgement is of form:

$$\Gamma; \Delta \setminus \Delta'; ((c : A_c \Leftrightarrow d : A_d) \setminus (c : A'_c \Leftrightarrow' d' : A'_d)); (\overline{q} \setminus \overline{q'}) \vdash s \ \mathbf{pid}$$
$$\Gamma; \Delta \setminus \cdot; ((c : A_c \Leftrightarrow d : A_d) \setminus (\uparrow \Leftrightarrow' \downarrow)); (\overline{q} \setminus \cdot) \vdash s \ \mathbf{pid}$$

where Γ and Δ are analogous to the ones in statement judgements and $\overline{q}$ is the message queue. Note that the second tracking context allows a different tracking client d'; this can occur from renaming $d' = d$. The former denotes a typical partial identity judgement, and the latter denotes a partial identity judgement after entering a termination state. Just as in the statement judgements, we maintain a notion of termination state using the up and down arrows $\uparrow$ and $\downarrow$. However, for stylistic reasons, we choose to use the tracking client $d : A_d$ to hold $\downarrow$ instead of Δ. Thus, Δ in a terminated state is simply empty: $\cdot$.

The first rule is a subsumption rule similar to the statement judgements where a terminated state can allow inference of any arbitrary state. In partial identity checks, a state consists of a tracking context, queue, and the linear context. As in the statement judgements, these will be encountered in control flow statements like if or while to allow non-terminated branches to continue if appropriate.

$$\frac{\Gamma; \Delta \setminus \cdot; (\omega \setminus (\uparrow \Leftrightarrow \downarrow)); (\overline{q} \setminus \cdot) \vdash s \ \mathbf{pid}}{\Gamma; \Delta \setminus \Delta'; (\omega \setminus \omega'); (\overline{q} \setminus \overline{q'}) \vdash s \ \mathbf{pid}} \ (\text{PID-TERM})$$

We begin with the partial identity judgement rules that do not involve using the tracking context. Thus, the following rules are almost identical to the statement judgements. The only glaring difference is that we now require statements that can modify variables not to modify any members of the queue. This makes sense because the queue only captures what variable it ought to send; if we allow these variables to be mutated, then partial identity no longer becomes guaranteed. A more involved check for mutation becomes necessary if we allow pointers to MCC0. In CC0, where we do have pointers, we simply disallow referencing for members of the queue.

$$\frac{\Gamma, x : \kappa \vdash e :: \kappa \qquad \mathbf{Pers}(x) \notin \overline{q}}{\Gamma, x : \kappa; \Delta \setminus \Delta; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash x = e \ \mathbf{pid}} \text{(PID-ASG-E)}$$

$$\frac{\Gamma, x : \tau \vdash e :: \kappa \qquad f : (\overline{\kappa}, \overline{A}) \to \tau \in \Sigma \qquad \mathbf{Pers}(x) \notin \overline{q} \qquad \overline{\mathbf{Ch}(d) \notin \overline{q}}}{\Gamma, x : \tau; \Delta, \overline{d : A} \setminus \Delta; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash x = f(\overline{e}, \overline{d}) \ \mathbf{pid}} \text{(PID-ASG-F)}$$

$$\frac{\Gamma \vdash e :: \mathbf{bool}}{\Gamma; \Delta \setminus \Delta; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash \mathbf{assert}(e) \ \mathbf{pid}} \text{(PID-ASSERT)}$$

$$\frac{\Gamma \vdash \overline{e} :: \overline{\tau} \qquad p : \mathbf{mon}((A, \overline{\tau}) \to A) \in \Sigma}{\Gamma; \Delta, d : A \setminus \Delta, d : A; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash \mathbf{mon}(d = p(d, \overline{e})) \ \mathbf{pid}} \text{(PID-MONITOR1)}$$

$$\frac{\Gamma \vdash \overline{e} :: \overline{\tau} \qquad p : \mathbf{mon}((A, \overline{\tau}) \to A) \in \Sigma}{\Gamma; \Delta \setminus \Delta; ((c : A' \Leftrightarrow d : A) \setminus (c : A' \Leftrightarrow d : A)); (\overline{q} \setminus \overline{q}) \vdash \mathbf{mon}(d = p(d, \overline{e})) \ \mathbf{pid}} \text{(PID-MONITOR2)}$$

$$\frac{\mathbf{Ch}(d') \notin \overline{q}}{\Gamma; \Delta, d' : A \setminus \Delta, d : A; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash d = d' \ \mathbf{pid}} \text{(PID-RENAME1)}$$

$$\frac{\Gamma \vdash \overline{e} :: \overline{\kappa} \qquad p : (\overline{\kappa}, \overline{A}) \to A' \in \Sigma \qquad \overline{\mathbf{Ch}(d) \notin \overline{q}}}{\Gamma; \Delta, \overline{d : A} \setminus \Delta, d' : A'; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash d' = p(\overline{e}, \overline{d}) \ \mathbf{pid}} \text{(PID-SPAWN1)}$$

$$\frac{\mathbf{Ch}(d) \notin \overline{q}}{\Gamma; \Delta, d : \{1\} \setminus \Delta; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash \mathbf{wait}(d) \ \mathbf{pid}} \text{(PID-WAIT1)}$$

$$\frac{\Gamma \vdash e :: \kappa \qquad \mathbf{Ch}(d) \notin \overline{q}}{\Gamma; \Delta, d : \{?\kappa; \overline{a}\} \setminus \Delta, d : \{\overline{a}\}; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash \mathbf{send}(d, e) \ \mathbf{pid}} \text{(PID-SEND-PERS1)}$$

$$\frac{\mathbf{Ch}(d') \notin \overline{q}}{\Gamma; \Delta, d : \{?A; \overline{a}\}, d' : A \setminus \Delta, d : \{\overline{a}\}; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash \mathbf{send}(d, d') \ \mathbf{pid}} \text{(PID-SEND-CH1)}$$

$$\frac{\mathbf{Pers}(x) \notin \overline{q} \qquad \mathbf{Ch}(d) \notin \overline{q}}{\Gamma, x : \kappa; \Delta, d : \{!\kappa; \overline{a}\} \setminus \Delta, d : \{\overline{a}\}; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash x = \mathbf{recv}(d) \ \mathbf{pid}} \text{(PID-RECV-PERS1)}$$

$$\frac{\mathbf{Ch}(d) \notin \overline{q}}{\Gamma; \Delta, d : \{!A; \overline{a}\} \setminus \Delta, d : \overline{a}, d' : A; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash d' = \mathbf{recv}(d) \ \mathbf{pid}} \text{(PID-RECV-CH1)}$$

$$\frac{\forall i \ (\Gamma; \Delta, d : A_i \setminus \Delta'; (\omega \setminus \omega'); (\overline{q} \setminus \overline{q'}) \vdash s_i \ \mathbf{pid}) \qquad \mathbf{Ch}(d) \notin \overline{q}}{\Gamma; \Delta, d : \{!\mathbf{choice}\{\overline{l, A}\}\} \setminus \Delta'; (\omega \setminus \omega'); (\overline{q} \setminus \overline{q'}) \vdash \mathbf{switch}(d) \ \{\overline{l : s}\} \ \mathbf{pid}} \text{(PID-SWITCH1)}$$

$$\frac{\Gamma; \Delta_1 \setminus \Delta_2; (\omega_1 \setminus \omega_2); (\overline{q_1} \setminus \overline{q_2}) \vdash s_1 \ \mathbf{pid} \qquad \Gamma; \Delta_2 \setminus \Delta_3; (\omega_2 \setminus \omega_3); (\overline{q_2} \setminus \overline{q_3}) \vdash s_2 \ \mathbf{pid}}{\Gamma; \Delta_1 \setminus \Delta_3; (\omega_1 \setminus \omega_3); (\overline{q_1} \setminus \overline{q_3}) \vdash s_1; s_2 \ \mathbf{pid}} \text{(PID-SEQ)}$$

$$\frac{\Gamma \vdash e :: \mathbf{bool} \qquad \Gamma; \Delta \setminus \Delta'; (\omega \setminus \omega'); (\overline{q} \setminus \overline{q'}) \vdash s_1 \ \mathbf{pid} \qquad \Gamma; \Delta \setminus \Delta'; (\omega \setminus \omega'); (\overline{q} \setminus \overline{q'}) \vdash s_2 \ \mathbf{pid}}{\Gamma; \Delta \setminus \Delta'; (\omega_1 \setminus \omega'); (\overline{q} \setminus \overline{q'}) \vdash \mathbf{if}(e) \ \{s_1\} \ \{s_2\} \ \mathbf{pid}} \text{(PID-IF)}$$

$$\frac{\Gamma\ \vdash e :: \textbf{bool} \qquad \Gamma; \Delta \setminus \Delta; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash s\ \textbf{pid}}{\Gamma; \Delta \setminus \Delta; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash s\ \textbf{pid}}\ \text{(PID-WHILE)}$$

$$\frac{\textbf{Pers}(h) \notin \overline{q}}{\Gamma, h : [\#; \overline{a}]; \Delta \setminus \Delta, d : \{\overline{a}\}; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash d = \textbf{acq}(h)\ \textbf{pid}}\ \text{(PID-ACQ)}$$

$$\frac{\textbf{Ch}(d) \notin \overline{q} \qquad \textbf{Pers}(h) \notin \overline{q}}{\Gamma, h : [\#; \overline{a}]; \Delta, d : \{\#; \overline{a}\} \setminus \Delta; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash h = \textbf{rel}(d)\ \textbf{pid}}\ \text{(PID-REL)}$$

The rules so far are for the most part identical to the statement judgements in section 2.5.3 since the tracking context and the queue were not used. The frequent $\notin$ clauses ensure that variables in the queue are not mutated. For persistent variables, this is reflected in assignments, but for linear channels, this is reflected in any statement that may progress a channel. The following rules consist of the more interesting parts of the partial identity check.

We first introduce the shift rule which allows message direction to change when the queue is empty.

$$\frac{\Gamma; \Delta \setminus \Delta; ((c : \{!t; \overline{a}\} \Leftarrow d : \{!t; \overline{a}\}) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s\ \textbf{pid}}{\Gamma; \Delta \setminus \Delta; ((c : \{!t; \overline{a}\} \Rightarrow d : \{!t; \overline{a}\}) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s\ \textbf{pid}}\ \text{(PID-SHIFT-RL)}$$

$$\frac{\Gamma; \Delta \setminus \Delta; ((c : \{?t; \overline{a}\} \Rightarrow d : \{?t; \overline{a}\}) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s\ \textbf{pid}}{\Gamma; \Delta \setminus \Delta; ((c : \{?t; \overline{a}\} \Leftarrow d : \{?t; \overline{a}\}) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s\ \textbf{pid}}\ \text{(PID-SHIFT-LR)}$$

where t denotes a generic entry in the session type, including any sendeable type or a choice (the ! specifies that it is an internal choice and so on), in which case the following $\overline{a}$ is empty, since an action cannot follow a choice.

We also add a shift rule to handle the unit type, which implicitly sends through the providing channel and therefore operates like a session type with $!t$:

$$\frac{\Gamma; \Delta \setminus \Delta; ((c : \{1\} \Leftarrow d : \{1\}) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s\ \textbf{pid}}{\Gamma; \Delta \setminus \Delta; ((c : \{1\} \Rightarrow d : \{1\}) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s\ \textbf{pid}}\ \text{(PID-SHIFT-RL2)}$$

A **wait** and **close** use the message **End**. Analogous to the statement judgement, **close** requires the linear context to be empty and the queue to be a singleton **End**, since no more messages can be sent after termination.

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((c : A \Leftarrow d : \{1\}) \setminus (c : A \Leftarrow \downarrow)); (\overline{q} \setminus \overline{q}, \textbf{End}) \vdash \textbf{wait}(d)\ \textbf{pid}}\ \text{(PID-WAIT2)}$$

$$\frac{}{\Gamma; \cdot \setminus \cdot; ((c : \{1\} \Leftarrow \downarrow) \setminus (\uparrow \Leftarrow \downarrow)); (\textbf{End} \setminus \cdot) \vdash \textbf{close}(c)\ \textbf{pid}}\ \text{(PID-CLOSE)}$$

recv and **send** record and consume **Pers**(x) and **Ch**(d') messages in the queue respectively.

$$\frac{\textbf{Pers}(x) \notin \overline{q}}{\Gamma, x : \kappa; \Delta \setminus \Delta; ((c : A \Leftarrow d : \{!\kappa; \overline{a}\}) \setminus (c : A \Leftarrow d : \{\overline{a}\})); (\overline{q} \setminus \overline{q}, \textbf{Pers}(x)) \vdash x = \textbf{recv}(d)\ \textbf{pid}}\ \text{(PID-RECV-PERS2)}$$

$$\frac{}{\Gamma; \Delta \setminus \Delta, d' : A; ((c : A \Leftarrow d : \{!A'; \overline{a}\}) \setminus (c : A \Leftarrow d : \{\overline{a}\})); (\overline{q} \setminus \overline{q}, \textbf{Ch}(d')) \vdash d' = \textbf{recv}(d)\ \textbf{pid}}\ \text{(PID-RECV-CH2)}$$

$$\frac{}{\Gamma, x : \kappa; \Delta \setminus \Delta; ((c : A \Rightarrow d : \{?\kappa; \overline{a}\}) \setminus (c : A \Rightarrow d : \{\overline{a}\})); (\textbf{Pers}(x), \overline{q} \setminus \overline{q}) \vdash \textbf{send}(d, x)\ \textbf{pid}}\ \text{(PID-SEND-PERS2)}$$

$$\frac{}{\Gamma; \Delta, d' : A' \setminus \Delta; ((c : A \Rightarrow d : \{?A'; \overline{a}\}) \setminus (c : A \Rightarrow d : \{\overline{a}\})); (\textbf{Ch}(d'), \overline{q} \setminus \overline{q}) \vdash \textbf{send}(d, d')\ \textbf{pid}}\ \text{(PID-SEND-CH2)}$$

Corresponding rules for the providing channel can be obtained by dualizing as usual:

$$\frac{\mathbf{Pers}(x) \notin \overline{q}}{\Gamma, x : \kappa; \Delta \setminus \Delta; ((c : \{?\kappa; \overline{a}\} \Rightarrow d : A) \setminus (c : \{\overline{a}\} \Rightarrow d : A)); (\overline{q} \setminus \overline{q}, \mathbf{Pers}(x)) \vdash x = \mathbf{recv}(c)\ \mathbf{pid}} \text{(PID-RECV-PERS3)}$$

$$\frac{}{\Gamma; \Delta \setminus \Delta, d' : A; ((c : \{?A'; \overline{a}\} \Rightarrow d : A) \setminus (c : \{\overline{a}\} \Rightarrow d : A)); (\overline{q} \setminus \overline{q}, \mathbf{Ch}(d')) \vdash d' = \mathbf{recv}(c)\ \mathbf{pid}} \text{(PID-RECV-CH3)}$$

$$\frac{}{\Gamma, x : \kappa; \Delta \setminus \Delta; ((c : \{!\kappa; \overline{a}\} \Leftarrow d : A) \setminus (c : \{\overline{a}\} \Leftarrow d : A)); (\mathbf{Pers}(x), \overline{q} \setminus \overline{q}) \vdash \mathbf{send}(c, x)\ \mathbf{pid}} \text{(PID-SEND-PERS3)}$$

$$\frac{}{\Gamma; \Delta, d' : A' \setminus \Delta; ((c : \{!A'; \overline{a}\} \Leftarrow d : A) \setminus (c : \{\overline{a}\} \Leftarrow d : A)); (\mathbf{Ch}(d'), \overline{q} \setminus \overline{q}) \vdash \mathbf{send}(c, d')\ \mathbf{pid}} \text{(PID-SEND-CH3)}$$

Switch statements and sending labels again follow a similar pattern:

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((c : A \Rightarrow d : \{?\mathbf{choice}\{\overline{l, A}\}\}) \setminus (c : A \Rightarrow d : A_i)); (\overline{q}, \mathbf{Label}(l_i) \setminus \overline{q}) \vdash d.l_i\ \mathbf{pid}} \text{(PID-LABEL2)}$$

$$\frac{\forall i\ (\Gamma; \Delta \setminus \Delta'; ((c : A \Leftarrow d : A_i) \setminus \omega'); (\overline{q}, \mathbf{Label}(l_i) \setminus \overline{q'}) \vdash s_i\ \mathbf{pid})}{\Gamma; \Delta \setminus \Delta'; ((c : A \Leftarrow d : \{!\mathbf{choice}\{\overline{l, A}\}\}) \setminus \omega'); (\overline{q} \setminus \overline{q'}) \vdash \mathbf{switch}(d)\ \{\overline{l : s}\}\ \mathbf{pid}} \text{(PID-SWITCH2)}$$

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((c : \{!\mathbf{choice}\{\overline{l, A}\}\} \Leftarrow d : A) \setminus (c : A_i \Leftarrow d : A)); (\overline{q}, \mathbf{Label}(l_i) \setminus \overline{q}) \vdash c.l_i\ \mathbf{pid}} \text{(PID-LABEL3)}$$

$$\frac{\forall i\ (\Gamma; \Delta \setminus \Delta'; ((c : \{!\mathbf{choice}\{\overline{l, A}\}\} \Rightarrow d : A) \setminus \omega'); (\overline{q}, \mathbf{Label}(l_i) \setminus \overline{q'}) \vdash s_i\ \mathbf{pid})}{\Gamma; \Delta \setminus \Delta'; ((c : \{!\mathbf{choice}\{\overline{l, A}\}\} \Rightarrow d : A) \setminus \omega'); (\overline{q} \setminus \overline{q'}) \vdash \mathbf{switch}(c)\ \{\overline{l : s}\}\ \mathbf{pid}} \text{(PID-SWITCH3)}$$

Forwarding the client simply renames the client in the tracking context, and forwarding the provider (with the client) requires the linear context and queue to be empty. As a trivial consequence of this, a monitor that immediately forwards is a valid partial identity monitor (though useless).

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((c : A \Leftrightarrow d : A') \setminus (c : A \Leftrightarrow d' : A')); (\overline{q} \setminus \overline{q}) \vdash d' = d\ \mathbf{pid}} \text{(PID-RENAME2)}$$

$$\frac{}{\Gamma; \cdot \setminus \cdot; ((c : A \Leftrightarrow d : A) \setminus (\uparrow \Leftrightarrow \downarrow)); (\cdot \setminus \cdot) \vdash c = d\ \mathbf{pid}} \text{(PID-FWD)}$$

Tailcalling on monitors is a special case to allow monitors on recursive session types.

$$\frac{\Gamma\ \vdash \overline{e} :: \overline{\tau} \qquad p : \mathbf{mon}((A, \overline{\tau}) \to A) \in \Sigma}{\Gamma; \cdot \setminus \cdot; ((c : A \Leftrightarrow d : A) \setminus (\uparrow \Leftrightarrow \downarrow)); (\cdot \setminus \cdot) \vdash c = p(d, \overline{e})\ \mathbf{pid}} \text{(PID-TAILCALL-MON)}$$

We reject spawning while consuming the client because it requires additional tracking states, adding more clutter to the already complicated inference rules. In particular, statements of form:

$$c = p(\dots, d, \dots)$$
$$d' = p(\dots, d, \dots)$$

are rejected (where c and d are the offering and client tracking channels respectively).

3.3.3 Declarations

Going back to the type judgements for declarations, declarations now contain monitors, which are somewhat analogous to **req**() and **ens**(). Monitors over clients (the inputs) can be mapped to equivalent statement level monitors, similar to **req**(). However, monitors over the providing channel require typechecking a wrapper process which we describe in section 3.3.4. Therefore, we will assume that declarations do not have any contracts, as in section 2.5.4.

$$\frac{\overline{x : \tau}; \cdot \setminus \downarrow; ((c : A \Leftarrow d : A) \setminus (\uparrow \Leftarrow \downarrow)); (\cdot \setminus \cdot) \vdash s \ \textbf{pid}}{\textbf{mon}(A\, c\, p(A\, d, \overline{\tau\, x})\{s\}) \ \textbf{decl}} \ \text{(DECL-LMON)}$$

Note that the initial direction of the tracking context does not matter; if the direction happened to be $\Rightarrow$, then the partial identity check will infer a valid shift since the queue is empty from (PID-SHIFT-LR).

3.3.4 Wrapper

So far, we have not formally accounted for contracts attached with functions or processes and instead explained that no specific judgement is required since they can be interpreted as syntactic sugar.

We first classify all contracts into two categories before and after. For ordinary contracts, this is precisely requires and ensures respectively and are already split. For monitors, before include all monitors on the arguments, whereas after include all monitors on the offering channel. Thus, no after monitor exists on a function since it does not offer any channel. A declaration then consists of four contracts (requires, ensures, before monitors, and after monitors). The before monitors and after monitors are of form:

$$\textbf{bmon}(d = p(d, \dots))$$
$$\textbf{amon}(c = p(c, \dots))$$

Note that for requires and ensures, the category names before and after indeed describe its semantics. However for monitors, since the direction of the messages are orthogonal to whether it is used as a client or provided, the names before and after only reflect a difference in how a translation ought to occur. Thus, although before and after monitors are generalizations of pre-condition and post-condition contracts respectively, the temporal implication behind the prefixes pre and post are lost in the generalization.

We introduce a set of translation rules that infers an appropriate wrapper for every declaration with contracts and converts them to the more primitive asserts and statement monitors. The rules assume that monitors are appropriately classified. In implementations, these rules can be applied before type checking to synthesize declarations; type checking these wrappers correspond to type checking the contracts on function level.

$$\frac{\tau\, f(\overline{\kappa\, x}, \overline{A\, d})\{ \begin{smallmatrix} \overline{\textbf{assert}(e_1)};\overline{\textbf{mon}(d{=}p(d,\overline{e_b}))}; \\ \tau result; result{=}f'(\overline{x},\overline{d}); \\ \overline{\textbf{assert}(e_2)};return\, result; \end{smallmatrix} \} \qquad \tau\, f'(\overline{\kappa\, x}, \overline{A\, d})\{s\}}{\tau\, f(\overline{\kappa\, x}, \overline{A\, d})[\overline{\textbf{req}(e_1)}, \overline{\textbf{ens}(e_2)}, \overline{\textbf{bmon}(d = p(d, \overline{e_b}))}, \cdot]\{s\}} \ \text{(WRAPPER-FUNC)}$$

where f' is a freshly generated name. Note that since functions cannot provide a channel, it cannot have any after monitors. Also, since f' is freshly generated, this means that a programmer cannot access f' directly – f' is only called through the original name f, so the declaration-level contracts are guaranteed to run in all invocations.

For processes, we allow after monitors. Since these monitors operate on the providing channel name, it is easier to rename the providing channel in the wrapper as well.

$$\frac{A'\,c'\,q(\overline{\kappa\,x},\overline{A\,d})\{\dfrac{\begin{array}{c}\overline{\mathbf{assert}(e_1)};\overline{\mathbf{mon}(d{=}p(d,\overline{e_b}))};\\ c{=}q'(\overline{x},\overline{d});\end{array}}{\begin{array}{c}\overline{\mathbf{assert}(e_2)};\overline{\mathbf{mon}(c{=}p(c,\overline{e_a}))};\\ c'{=}c\end{array}}\} \qquad A'\,c\,q'(\overline{\kappa\,x},\overline{A\,d})\{s\}}{A'\,c\,q(\overline{\kappa\,x},\overline{A\,d})[\overline{\mathbf{req}(e_1)},\overline{\mathbf{ens}(e_2)},\overline{\mathbf{bmon}(d=p(d,\overline{e_b}))},\overline{\mathbf{amon}(c=p(c,\overline{e_a}))}]\{s\}}\text{(WRAPPER-LPROC)}$$

We will assume that shared monitors exist (and hence, an after monitor for shared processes make sense) since it will be introduced in the following section.

$$\frac{S\,g'\,r(\overline{\kappa\,x})\{\dfrac{\begin{array}{c}\overline{\mathbf{assert}(e_1)};\overline{\mathbf{mon}(d{=}p(d,\overline{e_b}))};\\ S g;\, g{=}k'(\overline{x},\overline{d});\end{array}}{\begin{array}{c}\overline{\mathbf{assert}(e_2)};\overline{\mathbf{mon}(g{=}p(g,\overline{e_a}))};\\ g'{=}g\end{array}}\} \qquad S\,g\,r'(\overline{\kappa\,x})\{s\}}{S\,g\,r(\overline{\kappa\,x})[\overline{\mathbf{req}(e_1)},\overline{\mathbf{ens}(e_2)},\overline{\mathbf{bmon}(d=p(d,\overline{e_b}))},\overline{\mathbf{amon}(g=p(g,\overline{e_a}))}]\{s\}}\text{(WRAPPER-SPROC)}$$

As shown, the wrapper transforms all before contracts into statement contracts at the beginning, calls or spawns the original function or process, and then finally applies the after contracts. One key point is that the wrapper function should provide a different channel name such that after contracts can properly be called. Monitor declarations follow a similar pattern.

3.4 Examples

Listing 3: Trivial Monitors in CC0

```
<!int;> $c is_pos(<!int;> $d)
{
    int x = recv($d);
    assert(x > 0);
    send($c, x);
    $c = $d;
}

<!int;> $c is_pos2(<!int;> $d)
{
    int x = recv($d);
    assert(x > 0);
    wait($d);
    send($c, x);
    close($c);
}
```

Suppose the two processes are checked for partial identity. First, both processeses have a client $\$d$ and an offering $\$c$ of the same type. They have no other channels as arguments and are therefore possible candidates for being valid monitors.

Both processes begin similarly. On line 3 and 11, both processes bind the result of a receive on the client channel to x. At this point, the queue would consist of a single message $\mathbf{Pers}(x)$ with the message direction being $\Leftarrow$. Line 4 and 12, which although uses a variable x in the queue, is allowed because there is no assignment, so x cannot be mutated.

We now focus on is_pos; in line 5, we send x through $\$c$, which is valid since $\mathbf{Pers}(x)$ is the head of the queue. The forwarding at line 6 is valid because the queue is empty. At this point, we are done with the verification, so is_pos is indeed a valid monitor.

Moving on to the remaining statements in is_pos2, in line 13, we wait on $\$d$. At this point,

$$\overline{q} = \mathbf{Pers}(x), \mathbf{End}$$

Line 14 is therefore valid because $\mathbf{Pers}(x)$ is at the head of the queue, and similarly, line 15 is valid because $\mathbf{End}$ is now the only queue member. Therefore, is_pos2 is also a valid monitor.

Listing 4: List Monitor in CC0

```
choice list {
    <> Nil;
    <!int; !choice list> Cons;
};
typedef <!choice list> list;

list $c nil() {
    $c.Nil;
    close($c);
}

list $c cons(int n, list $d) {
    $c.Cons;
    send($c, n);
    $c = $d;
}

list $c allbigger_than(list $d, int threshold) {
    switch ($d) {
    case Nil:
        wait($d);
        $c.Nil;
        close($c);
    case Cons: {
        int res  = recv($d);
        assert(res > threshold);
27         $c.Cons;
28         send($c, res);
29         $c = allbigger_than($d, threshold);
30     }
31     }
32 }
33 
34 int sum(list $d)
35 //@monitor $d = allbigger_than($d, 10);
36 {
37     switch ($d) {
38     case Nil: { wait($d); return 0; }
39     case Cons: {
40         int n = recv($d);
41         return n + sum($d);
42     }
43     }
44 }
45 
46 int main()
47 {
48     list $a = nil();
49     $a = cons(11, $a);
50     $a = cons(14, $a);
51     $a = cons(12, $a);
52     int b = sum($a);
53     return 0;
54 }
```

Here, we introduce an entire program. Since line 35 uses the process allbigger_than as a monitor, the typechecker infers that it must be checked using partial identity. Again, we will briefly walk through the partial identity checking.

list is a recursive session type: $\{!choice\{(Nil, \{1\}), (Cons, \{!int, list\})\}\}$. We first case over the choices in line 19. In line 20, we assume that we receive the label Nil from $\$d$, so the queue consists of $\textbf{Label}(Nil)$. Lines 21 to 23 are trivial.

In the Cons branch from line 24, we first bind the result of a receive to res. At this point, $\overline{q} = \textbf{Label}(Cons), \textbf{Pers}(res)$. In line 27, we send the label Cons, which is allowed because the head of the queue is $\textbf{Label}(Cons)$. Similarly, we send res in line 28. By line 29, the queue is empty, so we are allowed to recurse using allbigger_than, which is a monitor.

4 Shared Monitors

We next introduce shared monitors. Similar to linear monitors, we view shared monitors as shared processes that satisfy an extended notion of partial identity. See 4 for a graphical representation of a typical shared monitor situation.

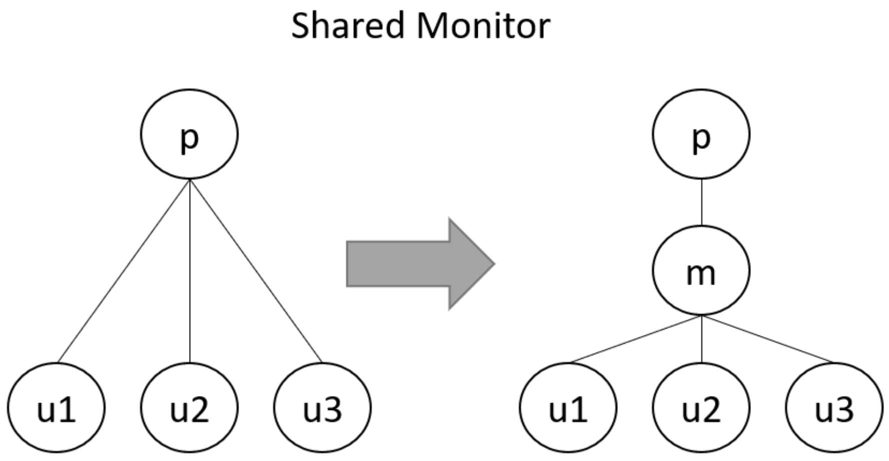

Figure 4: A shared process p being monitored by m. Note that although p now only has one client, it is still a shared process.

As it turns out, partial identity for shared monitors at first seems fairly intuitive; similar to the recurring idea that a receive must record to a queue and send must consume a queue, we can view operations on shared channels as variants of receiving and sending, which we will formulate later.

However, we found that a naive code transformation of an otherwise valid monitor statement will not give intended results and therefore require a more involved transformation step. Hence, unlike for linear monitors, where partial identity and type checking were the main interest whereas the translation step was a mere side note, in this section, the translation step is the main interest whereas the type checking is quite trivial. Nevertheless, we begin with the typechecking.

$$
\begin{aligned}
\mathit{decl} &::= \ldots \\
&\mid \mathbf{mon}(S\, g\, r(Sh, \overline{tp\, x})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\}) \\
s &::= \ldots \\
&\mid \mathbf{mon}(g = p(h, \overline{e})) \\
\mathit{sigtp} &::= \ldots \\
&\mid \mathbf{mon}((S, \overline{tp}) \to S)
\end{aligned}
$$

The corresponding type rule for statement level shared monitors is reminiscent of the linear version.

$$\frac{\Gamma \vdash \overline{e} :: \overline{\tau} \qquad k : \mathbf{mon}((S, \overline{\tau}) \rightarrow S) \in \Sigma}{\Gamma, h : S \; ; \; \Delta \setminus \Delta \vdash \mathbf{mon}(h = k(h, \overline{e})) :: \Theta} \text{(STM-SMONITOR)}$$

Since shared statement monitors are analogous to the linear ones, we opt not to separate the two variants. This does mean however that rules where we assumed generic linear monitors can be generalized to shared or linear – in these cases, some uses of suggestive variables such as d or p should be appropriately reconsidered. For example, monitors over functions and process now possibly include shared monitors, and so on.

4.1 Type Judgements

4.1.1 Declaration

$$\frac{\overline{x : \tau}; \cdot \setminus \downarrow; ((g : S \Leftarrow h : S) \setminus (\uparrow \Leftarrow \downarrow)); (\cdot \setminus \cdot) \vdash s \; \mathbf{pid}}{S \, g \, r(S \, h, \overline{\tau \, x})\{s\} \; \mathbf{decl}} \text{(DECL-SMON)}$$

4.1.2 Statement

We want to reiterate the idea that monitors are effectively an expansion over the type structure of the tracking channels augmented with additional local computation and assertions. Partial identity checking for linear channels were complicated mostly because of the rich structure of linear channels with many available expansion operations. However for shared channels, we can only acquire/release for the client and accept/detach for the provider. For ordinary statement judgements, we observed that after acquiring or accepting, the type checking was continued in the approriate linear setting where it was assumed that either the linear channel terminated or was upshifted back appropriately. A similar idea applies for partial identity; any non-trivial shared monitor consists of accept and acquire, a linear section, and terminate with release and detach. Thus, checking a shared monitor at first seems to be a matter of checking for appropriate form and then checking the middle linear section, meaning we can defer much of the work to the partial identity for linear channels that we introduced in the previous section.

When we view shared operators as messages, an accept is effectively a receiving of a downshift message, and a detach is a sending of an upshift message. By duality, we then have that an acquire is sending a downshift while release is sending an upshift. Thus, we append two additional possible messages in our queue.

$$q \; ::= \; \ldots |\mathbf{Dshift}|\mathbf{Ushift}$$

We first introduce the two shift rules. Surprisingly, both rules anticipate a right direction; an accept is interpreted as a message from the user, and we will later explain why it makes sense for a release to anticipate a message from the user as well.

$$\frac{\Gamma; \Delta \setminus \Delta; ((g : [\#; \overline{a}] \Rightarrow h : [\#; \overline{a}]) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s \ \mathbf{pid}}{\Gamma; \Delta \setminus \Delta; ((g : [\#; \overline{a}] \Leftarrow h : [\#; \overline{a}]) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s \ \mathbf{pid}} \ (\text{PID-SHIFTD})$$

$$\frac{\Gamma; \Delta \setminus \Delta; ((c : \{\#; \overline{a}\} \Rightarrow d : \{\#; \overline{a}\}) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s \ \mathbf{pid}}{\Gamma; \Delta \setminus \Delta; ((c : \{\#; \overline{a}\} \Leftarrow d : \{\#; \overline{a}\}) \setminus \omega); (\cdot \setminus \overline{q}) \vdash s \ \mathbf{pid}} \ (\text{PID-SHIFTU})$$

A monitor involving shared channels consists of:

1. accept
2. acquire
3. linear portion
4. release
5. detach

where well-typed arbitrary local computation (statements that do not progress or consume the queue) can be inserted anywhere. Accept must precede acquire since otherwise, a shared monitor will acquire a shared resource and hold it, which can violate partial identity. The following rules are then reflections of this accept, acquire, linear, release, and finally detach paradigm.

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((g : [\#; \overline{a}] \Rightarrow h : [\#; \overline{a}]) \setminus (c : \{\overline{a}\} \Rightarrow h : [\#; \overline{a}])); (\cdot \setminus \mathbf{Ushift}) \vdash c = \mathbf{acc}(g) \ \mathbf{pid}} \ (\text{PID-ACC})$$

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((c : \{\overline{a}\} \Rightarrow h : [\#; \overline{a}]) \setminus (c : \{\overline{a}\} \Leftarrow d : \{\overline{a}\})); (\mathbf{Ushift} \setminus \cdot) \vdash d = \mathbf{acq}(h) \ \mathbf{pid}} \ (\text{PID-ACQ2})$$

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((c : \{\#; \overline{a}\} \Rightarrow d : \{\#; \overline{a}\}) \setminus (c : \{\#; \overline{a}\} \Leftarrow h : [\#; \overline{a}])); (\cdot \setminus \mathbf{Dshift}) \vdash h = \mathbf{rel}(d) \ \mathbf{pid}} \ (\text{PID-REL2})$$

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((c : \{\#; \overline{a}\} \Leftarrow h : [\#; \overline{a}]) \setminus (g : [\#; \overline{a}] \Leftarrow h : [\#; \overline{a}])); (\mathbf{Dshift} \setminus \cdot) \vdash g = \mathbf{det}(c) \ \mathbf{pid}} \ (\text{PID-DET})$$

4.2 Acknowledgement of Accept/Acquire

Unfortunately, the model of accept, acquire, linear code, release, and then detach is not sufficient for partial identity. This issue can be intuitively understood when viewing shared processes as instead shared resources. With this viewpoint, a client succesfully acquiring a shared resource puts itself into a critical region. However, if the shared resource is monitored, we get an arbitrarily long window of delay between the accept and acquire; this can most notably occur when some client has an unmonitored access to the resource, which can be intentionally created through aliasing. Thus, a client succesfully acquiring a monitored shared resource does not necessarily mean that it has actually acquired the shared resource, and it is not a mere timing issue.

To resolve this issue, we first introduce an acknowledgement of acceptance to be sent by the provider whenever it accepts. In implementation, we append each shift by a **!bool**. The two-valued-ness of booleans are irrelevant; the same effect can be achieved through a traditional unit type. However, we choose not to do so to keep the language minimal but also to avoid confusion between the unit action 1 which signals termination as we introduced and the unit type 1 which is an ordinary type with only one value. In general, the session types

$$\{\#; \overline{a}\}$$

$$[\#; \overline{a}]$$

should respectively transform to

$$\{\#; !\mathbf{bool}; \overline{a}\}$$

$$[\#; !\mathbf{bool}; \overline{a}]$$

Then, outside a monitor, a statement of form

$$c = \mathbf{acc}(g)$$

should transform to

$$c = \mathbf{acc}(g); \mathbf{send}(c, \mathbf{true})$$

and similarly, a statement of form

$$d = \mathbf{acq}(h)$$

should transform to

$$d = \mathbf{acq}(h); x = \mathbf{recv}(d)$$

where x is fresh and therefore not referenced in the context at which this transformation occurs.

Inside a monitor with providing shared channel g and client h that is already checked for partial identity, we encounter the accept and acquire paradigm in the following format:

$$c = \mathbf{acc}(g); s_1; d = \mathbf{acq}(h); s_2$$

where s_1 is some local well-typed computation that does not affect the tracking context nor the queue. These sequences of statements should be transformed to

$$c = \mathbf{acc}(g); s_1; d = \mathbf{acq}(h); x = \mathbf{recv}(d); \mathbf{send}(c, x); s_2$$

Why do these transformations solve the problem? Going back to the original scenario where a client acquires a monitored shared resource, we now see that the client is then blocked from proceeding while waiting for an acknowledgement $x = \mathbf{recv}(d)$. Looking into the monitor, we see that the monitor sends an acknowledgement only after it has succesfully acquired the resource.

As with the other introduced notions in the paper, we do not prove a desirable property that a well-typed program will remain well-typed under this global transformation. However, we hope that it is very intuitive why that would be the case; we modify all shared session types and its subsequent uses to add an acknowledge, making sure no non-monitors that interact with the monitors do not have mismatched session types. For monitors, we make sure to receive an acknowledge and then forward the result, maintaining partial identity as the queue must be empty at the point of receiving the acknolowdgement..

4.3 To Release Message

Unfortunately, the acknowledgement change is a step in the right direction but is still not sufficient. Again, consider a client accessing an unmonitored shared resource. After succesfully acquiring the resource, the client enters a critical region and then exits the critical region after it has released the resource. Now consider a monitored shared resource. With the acknowledgement changes, we have succesfully fixed the issue where the client can enter a critical region before actually acquiring the resource. However, the monitor, can release the resource before the client has actually released the resource as long as the protocol allows for it. Thus, we still have an issue where the client can unknowingly and prematurally exit a critical region by the whim of the monitor.

This issue is again resolved through a so called "to release" message that is sent by the client. We first prepend all shifts with a ?**bool** (again, we do not use the value of the boolean), meaning before an upshift, a boolean must be sent. Thus, we must yet again apply a global transformation on all non-monitors.

$$h = \mathbf{rel}(d)$$

transforms to

$$\mathbf{send}(d, \mathbf{true}); h = \mathbf{rel}(d)$$

and

$$g = \mathbf{det}(c)$$

transforms to

$$x = \mathbf{recv}(c); g = \mathbf{det}(c)$$

where again, x is fresh.

Inside a monitor with providing shared channel g and client h that is already checked for partial identity, we encounter the release and detach paradigm in the following format:

$$h = \mathbf{rel}(d); s_1; g = \mathbf{det}(c); s_2$$

where s_1 is some local well-typed computation that does not affect the tracking context nor the queue. These sequences of statements should be transformed to

$$x = \mathbf{recv}(c); \mathbf{send}(h, x); h = \mathbf{rel}(d); s_1; g = \mathbf{det}(c); s_2$$

Going back to the issue we mentioned where, we see that the monitor will now be blocked from releasing the resource until signaled by the client. Thus, we have resolved the problem that the monitor can cause the client to unknowingly exit a critical region by prematurally releasing the resource. Again, we do not prove the desirable property that this transformation still leads to a well-typed program.

Remark We see that enabling contracts for shared processes suddenly require global-level transformations to the code. Until now, all code transformations were local. These transformations were required because we allow both monitored and unmonitored access to shared channels through aliasing. For example, consider some shared channel h. We can then alias h: $h' = h$. Next, if we monitor h only: **mon**($h = mon(h, \cdots)$), then we have h' giving unmonitored access to the original resource and h giving monitored access. In these situations, we must somehow instrument the program such that acquiring a monitored resource is equivalent to acquiring the resource itself and so on, which due to the semantics of manifest sharing requires additional care in the form of these acknowledgement and "to release" messages.

4.4 Example

Consider an implementation of a shared integer. It consists of a shared process where after a downshift, it becomes an external choice consisting of the labels *get* and *set* where if *get* is selected by a user, we send an integer and then upshift back to the original type, and if *set* is selected by a user, we receieve an integer to update the locally stored integer and then upshift back to the original type. The type declarations of this system written in CC0 is introduced in listing 5.

Listing 5: Type Declarations for a shared integer.

```
typedef <?choice shd_prov> A;
typedef <#; A> S;
choice shd_prov {
    <?int; S> Set;
    <!int; S> Get;
};
```

The full code is listed in the appendix (A) and the more general implementation of shared variables is discussed in detail in section 5.2.1.

In this section, we detail the partial identity check for a sample monitor which checks that set messages are monotonically increasing.

Listing 6: Monitor code that enforces monotonicity on a shared integer.

```
// checks if Set messages are monotonically increasing
S #c mon(S #d, int prev)
{
    A $c = (A)#c; // accept
    A $d = (A)#d; // acquire
    switch($c)
    {
        case Set: {
            int n = recv($c);
            assert(n >= prev);
            prev = n;
            $d.Set;
            send($d, n);
            break;
15         }
16         case Get: {
17             $d.Get;
18             int n = recv($d);
19             send($c, n);
20             break;
21         }
22     }
23     #d = (S)$d; // release
24     #c = (S)$c; // detach
25     #c = mon(#d, prev);
26 }
```

At lines 11 and 12, we see the only possible way shared monitors can begin (aside from local computations) – an accept followed by an acquire. At this point, our queue is empty and our tracking context consists of the linear channels

$$(\$c : A \Leftarrow \$d : A)$$

with A as defined in listing 5.

Therefore, lines 13 to 29 corresponds to the linear section, where we follow linear partial identity checking. Although we already showcase some partial identity examples in section 3.4, we will still briefly discuss the linear partial identity check that occurs inside lines 13 to 29.

The switch statement at line 13 pattern matches over the external choice of shd_prov. Since there are two labels, there are correspondingly two cases in the switch statement.

At line 15, we assume we receive a Set label, meaning $\$c$ is now of type $\{?int; S\}$ and the queue consists of the label message $\textbf{Label}(Set)$. We then receive an int from $\$c$, appending $\textbf{Pers}(n)$ to the queue. At lines 19 and 20, we clear the queue by first sending the label Set to $\$d$ and then n. At this point, we finish the Set branch with an empty queue and both $\$c$ and $\$d$ ready to upshift.

For the other branch from line 16, we assume a Get label. We immediately clear the queue by sending Get to $\$d$ at line 24, receiving an integer from $\$d$ at line 25, and clearing the queue at line 26 by sending n to $\$c$. Similar to the Set branch, we break with an empty queue and both $\$c$ and $\$d$ ready to upshift, making the entire switch statement valid.

At lines 30 and 31, we detach and release. Finally, we recurse with a tailcall, which is handled by a shared monitor extended version of (PID-TAILCALL-MON).

5 Case Studies

5.1 Linear Monitors in CC0

5.1.1 Service Verification

A typical service that requires input from the client can be modelled using a linear session type. For example, consider a service that when given an integer greater than 1, returns the prime factorization of it as a list. There are two natural ways to express the protocol of such a service. In both cases, we initially begin with an input of an int. Next, the service can either continue as a list or instead send a list, which itself is a channel, and then terminate. We choose the former for no particular reason. See listing 7 for the complete implementation of the monitor.

A monitor of such a service can first verify that the initial input is indeed a natural number. This requirement is captured in line TODO. Next, the monitor can verify that the provided list of factors indeed multiply to the original number. This is done by a local variable y that is multiplied everytime the monitor receives a factor as shown in line 32. When the monitor receives a Nil, signifying that no more factors exist, then y is required to match the original input x in line 24. The actual primality of the factors are intentionally left unverified as a design choice since that seems to be just as hard a problem as the service that we are monitoring. Technically, it is possible to verify that each p at line 31 is prime by locally running a factorization algorithm. A probabilistic test, for example Fermat's primality test, can instead sensibily be added, which would allow the monitor to claim great confidence that the factors are indeed prime factors.

Listing 7: An implementation of a service verification monitor.

```
1  choice list {
2      <> Nil;
3      <!int; !choice list> Cons;
4  };
5  typedef <!choice list> list;
6  typedef <?int; !choice list> serv;
7
8  // verifies that the initial input is a natural number
9  // and that the provided list multiplies to the original input
10 // importantly, it does not check if the list is all prime
11 serv $c ver_factor(serv $d)
12 {
13     int x = recv($c);
14     //@assert x > 0; // the service requires a natural number
15     send($d, x);
16     // at this point, both $c and $d are lists
17     int y = 1; // cumulative product of the factors
18     while(true)
19     {
20         switch($d)
21         {
22             case Nil:
```

```
            {
                //@assert y == x;
                $c.Nil;
                wait($d); close($c);
            }
            case Cons:
            {
                $c.Cons;
                int p = recv($d);
                y = y * p;
                send($c, p);
            }
        }
    }
}

serv $c factor()
//@monitor $c = ver_factor($c);
{
    // ...
}
```

Although the shown service has a relatively simple protocol, this structure of monitoring can readily apply to more involved protocols.

5.1.2 Binary Search Tree

On the other hand, processes can be used to model complex data structures such as a binary search tree.

Consider a binary search tree using a split join model where each linear process represents a node in the tree. The session type representing a binary search tree process is shown in listing 8.

Listing 8: Session type representing a binary search tree.

```
typedef <?choice bst> bst;
choice bst {
    <?int; !bst; !bool; !bst; > split;
    <?bst; bst>                 join;
    <?int; ?int; bst>           chk;
};
```

A binary search tree is an external choice with three labels split, join, and chk. After receiving the split label, the process receives an int (denoting the node at which the process should split at) and then proceeds to split the tree and send the left binary search tree, a boolean indicating whether the node was found or not, and then the right binary search tree. After receiving a join label, the process receives a binary search tree and then remains as a binary search tree (where the implication is that the received binary search tree was merged with the current process). Finally, after receiving a chk label, the process receives two integers representing the expected range of itself and its children and remains itself. While the split and join labels are required for implementation, the chk label is a proposition of some form of non-monitor contract which we discuss in this section.

Let us consider a possible monitor (hence independent of the label chk) mon as shown in 9.

Listing 9: Proposal of a monitor.

```
 1 bst $c mon(bst $d, int lo, int hi)
 2 {
 3     switch($c)
 4     {
 5     case split:
 6     {
 7         int k = recv($c);
 8         $d.split;
 9         send($d, k);
10         bst $l2 = recv($d);
11         bool b = recv($d);
12         bst $r2 = recv($d);
13         wait($d);
14         assert(!b || (lo < k && k < hi));
15         send($c, $l2);
16         send($c, b);
17         send($c, $r2);
18         close($c);
19     }
20     case join:
21     {
22         bst $t = recv($c);
23         //@monitor $t = mon($t, hi, 0x7FFFFFFF);
24         $d.join;
25         send($d, $t);
26         $c = mon($d, lo, 0x7FFFFFFF);
27     }
28     case chk:
29     {
30         $d.chk;
31         int l = recv($c);
32         int u = recv($c);
33         send($d, l);
34         send($d, u);
35         $c = mon($d, lo, hi);
36     }
37     }
38 }
```

Ignoring the major details of implementation, the monitor sits between two nodes and locally store a constraint on the range of the values. These values are compared with a split request at line 21 with the meaning that whenever a split request is propagated to this monitor, then it must be in the range (otherwise, the tree should have been split somewhere in the ancestor).

This style of monitoring seems to be the best effort one can make at attempting to enforce a kind of global invariant that a binary search tree must be appropriately ordered across multiple interconnected session types. On the other hand, checking for this invariant is simple through the chk label. Listing 10 showcases a node's implementation of such a label, which essentially propagates the check to its children while it updates the bound.

Listing 10: Chk case for the node implementation.

```
    case chk:
    {
        int lower = recv($t);
        int upper = recv($t);
        assert(lower <= k && k <= upper);
        $l.chk; // left side's new bound: [lower, k]
        send($l, lower);
        send($l, k);
        $r.chk; // right side's new bound: [k, upper]
        send($r, k);
        send($r, upper);
        $t = node($l, $r, k);
    }
```

However, this is clearly not a monitor and is instead dynamic checking that is coupled with the implementation of a node. Given our theory so far, it is not clear how one can even statically check that this *chk* label is independent to the implementation (that is, somehow removing it will still give a partially equivalent program). Thus, we see a limitation of monitors; since they can only sit in between two nodes and cannot communicate with other nodes, it does not seem to be able to express certain global-level invariants in interconnected systems. The fact that there are seemingly useful contracts one can write that is not expressable as monitors suggests that monitors lack expressiveness over ordinary contracts.

5.2 Shared Monitors in CC0

5.2.1 Shared Ordinary Variables

In section 4.4, we briefly introduced the shared session type representing a process that offers a shared integer. Although the full code is available in A, we introduce a shortened version of the code without the monitor implementation in 11.

Listing 11: An implementation of a shared integer.

```
typedef <?choice shd_prov> A;
typedef <#; A> S;
choice shd_prov {
    <?int; S> Set;
    <!int; S> Get;
};

S #c shd_int()
{
    int n = 0; // default
    while (true)
    {
        A $c = (A)#c; // accept
        switch($c)
        {
            case Set: {
                n = recv($c);
                break;
            }
            case Get: {
                send($c, n);
                break;
            }
        }
        #c = (S)$c; // detach
    }
}
```

The process definition is fairly self-explanatory; when a Set label is received, it sets the internal variable n, and when a Get label is received, it sends the internal variable n.

Although CC0 does not implement parametric polymorphism [Rey83], it is easy to imagine that shared variables for any ordinary types (anything with a notion of value) except pointers can be arbitrarily constructed with this paradigm. Using a shared monitor (similar to the example in A), we can then apply predicates to the received value whenever a Set label is received. Hence, when we implement shared variables as shared processes, shared monitors then become a natural way to refine the underlying variable type [Pfe95]. This parallels the idea that refinement types of linear sessions can be implemented as monitors as introduced by Gommerstadt [Gom19b].

5.2.2 Meta Protocol Enforcement

We next investigate a CC0 implementation by Pfenning [Pfe19] of a centralized form of Mitchell and Merritt's distributed deadlock detection algorithm [MM84]. The algorithm assumes a distributed system with shared resources (implemented with shared session types) and nodes. From a node's perspective, this implementation consists of a global shared process that acts as an overseer to some distributed system with all involving nodes and resources known ahead of time. The global overseer can detect (and with some modifications even recover) from deadlocks assuming that each node adheres to not only the protocol denoted by the session type shown in 12 but also a sort of "meta protocol" across multiple acquire release cycles.

Listing 12: The protocol for the global overseer.

```
typedef <?choice overseer> loverseer;
typedef <# ; loverseer> soverseer;

choice overseer {
  <?int ; ?int ; soverseer> tryacq;  //pid, rid, upshift
  <?int ; ?int ; soverseer> didacq;  //pid, rid, upshift
  <?int ; ?int ; soverseer> willrel; //pid, rid, upshift
};
```

To illustrate the meta protocol, we consider a situation where some node with an id p is acquiring a shared resource $\#res$ with an id r. Because the aforementioned overseer is global, it is also available through some channel s of type soverseer. When the node is ready to acquire the shared resource, it must first notify the global overseer before attempting the acquire through the tryacq choice. After the acquiring was succesful, the node must then notify the global overseer that the acquire was succesful through the didacq choice. Later, when the node is ready to release the shared resource, it can then notify the global overseer that it will release the resource before the actual release with the willrel choice. A minimum working example of such a process is showcased in listing 13.

Listing 13: A simple example showcasing a simple node adhering to the meta protocol.

```
<> $c node(int p, int r, srestp #sres, soverseer #sa)
{
    loverseer $la = (loverseer)#sa; // acquire #sa
    $la.tryacq; // notify the overseer that an attempt at acquire will begin
    send($la, p); send($la, r);
    #sa = (soverseer)$la; // release $la
    lrestp $lres = (lrestp)#sres; // the actual attempt at acquiring the resource
    // code procedes once #sres has been acquired

    loverseer $la = (loverseer)#sa; // acquire #sa
    $la.didacq; // notify the overseer that the previous acquire was completed
    send($la, p); send($la, r);
    #sa = (soverseer)$la; // release $la

    // critical region code using $lres

    loverseer $la = (loverseer)#sa; // acquire #sa
    $la.willrel; // notify the overseer that the resource will now be released
    send($la, p); send($la, r);
    #sa = (soverseer)$la; // release $la

    #sres = (srestp)$lres; // the actual release of the resource

    close($c);
}
```

As shown in this simple case, a node must send three notifications through three individual acquire release cycles for every shared resource it acquires. For some unique resource r, a node must send a tryacq, followed by a didacq, and finally a willrel for that particular r.

What if a node requires ownership of multiple shared resources at the same time? In this case, the final willrel notification can be delayed according to the system. However, a tryacq must always be followed be a didacq. Therefore, we have some notion of a meta protocol for this shared service – for a fixed node, we must begin with a tryacq followed by a didacq. At this point, we can either perform yet another tryacq followed by a didacq or instead send a willrel. If we sent a willrel, we can again either send a tryacq followed by a didacq or yet another willrel. See figure 5 for a graphical representation of this meta protocol.

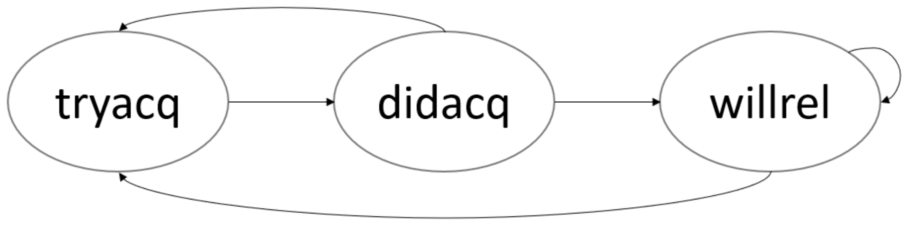

Figure 5: The meta protocol that a fixed node is required to follow across multiple acquire release cycles.

Because shared session types cannot remember any information about a previous acquire release cycle, this sort of meta protocol cannot be statically enforced using our typesystem. Furthermore, this issue is worsened by the fact that this global overseer can be acquired and released by multiple processes, and since the meta protocol only applies to a fixed node, simply observing that a tryacq was sent twice in a row is not sufficient since two separate nodes could have sent a tryacq. However as it turns out, shared monitors can enforce this meta protocol by locally keeping track of each process's last sent notification and upon receiving another notification, verifying that it is indeed an allowed message according to the meta protocol.

Aside from this meta protocol, a monitor can verify that a tryacq and a subsequent didacq were indeed on the same resources (it makes no sense for a node to claim to be acquiring some resource r but then immediately after notify that it has acquired some other resource r'). It can also check that it is not claiming to release a resource it never acquired. Finally, it can also check that it is not attempting to acquire a resource it currently owns.

We layout the structure of the monitor with the helper functions in listing 14. The monitor, named verify_protocol, keeps a local list of process statuses, where the type definition of a process status is shown in line 2. Some helper functions interacting with the local data structure are forward declared.

Listing 14: Monitor structure for the global overseer.

```
 1 typedef struct process_status process_status;
 2 struct process_status {
 3     int pid;  // process id
 4     bool acquiring; // is it in the middle of acquiring?
 5     int acquiring_res; // if acquiring, should hold the resource id that is being acquired
 6     int_list* acquired_res; // list of acquired resources
 7 };
 8 
 9 typedef struct pstatus_list pstatus_list;
10 struct pstatus_list {
11     process_status* ps;
12     pstatus_list* next;
13 };
14 
15 process_status* new_process_status(int p);
16 
17 // true if list contains r
18 // should be called when will_release is received
19 bool found_res(int_list* acquired_res, int r);
20 
21 // removes the first occurance of x from int_list
22 int_list* remove_res(int_list* acquired_res, int x);
23 
24 // returns pointer to a process status corresponding to p
25 // if it cannot find it, returns a null pointer
26 process_status* find(pstatus_list* pss, int p);
27 
28 // should initially be attached with a null pointer
29 soverseer #sa verify_protocol(soverseer #sb, pstatus_list* pss)
30 {
31     loverseer $la = (loverseer)#sa; //acc
32     loverseer $lb = (loverseer)#sb; //acq
33     switch ($la) {
34         case tryacq: {
35             ...
36         }
37         case didacq: {
38             ...
39         }
40         case willrel: {
41             ...
42         }
43     }
44     #sb = (soverseer)$lb;
45     #sa = (soverseer)$la;
46     #sa = verify_protocol(#sb, pss);
47 }
```

The implementations of the three individual cases (lines 35, 38, and 41) share redundant code and so in the interest of space, we show a sample implementation of the didacq case in listing 15.

Listing 15: Implementation of the didacq case.

```
1  case didacq: {
2      int p = recv($la);
3      int r = recv($la);
4      process_status* ps = find(pss, p);
5      assert(ps != NULL); // ps should exist since tryacq must have been called
6      assert(ps->acquiring);
7      assert(ps->acquiring_res == r);
8      int_list* new_acq_res = alloc(struct int_list);
9      new_acq_res->n = r;
10     new_acq_res->next = ps->acquired_res;
11     ps->acquired_res = new_acq_res;
12     ps->acquiring = false;
13     $lb.didacq;
14     send($lb, p);
15     send($lb, r);
16     break;
17 }
```

In line 4, we search for the process status entry matching p, the process id we receieve. The following three lines at lines 5 to 7 assert that the process status entry exists, a tryacq was previously called, and the previous tryacq call was on the same resource as r. At this point, we store the resource id r as a resource that is already owned by p to later be checked during willrel to verify the requirement that willrel must only be communicated on a resource the process owns. At line 12, we update our status to no longer acquiring. This means that a next iteration with the same process allows either a tryacq or a willrel.

The full implementation of this monitor is available in the appendix (C).

Although the verifications are useful and interesting in their own right, we particularly found that this idea that monitors can verify these meta protocols that essentially require fixed clients to follow some protocol across multiple acquire release cycles to be a very interesting part about shared monitors. This idea of a meta protocol seems to appear in many applications of shared channels, and so we can conclude that monitors can naturally check these meta protocols that cannot be statically typechecked.

We provide no formal description of how this meta protocol monitor can be implemented, but informally, a general implementation would involve a global transformation where some message indicating a unique identifier (say, an int) is inserted to all shared session types (and their corresponding linear session types) following a downshift. This id will then be used by the meta protocol monitor to locally construct some map of the protocols per unique id.

6 Conclusion

In this thesis, we have introduced groundwork on the implementation of linear session types and their corresponding known model of contracts, (linear) monitors, to an imperative language. We further extend the typesystem with shared session types following the manifest sharing semantics where we extended linear monitors to its manifest shared variant and postulated its type rules and semantics. We also enriched the typesystem of CC0 and its compiler with these additions, which we used to review some practical and theoretical cases of these monitors. Some key observations that we would like to highlight include observations that linear monitors seem to lack expressiveness in certain ways (5.1.2), shared monitors being able to express refinement of shared types (5.2.1), and shared monitors seem to be capable of expressing meta protocols, which seem to appear in many uses of shared processes.

An obvious future work is to prove many of the claims we have made. In particular, although we intuitively believe that our proposed type rules and semantics for shared monitors indeed capture a shared extension to the linear monitors, it seems that a partial identity proof for shared monitors analogous to the proof for linear monitors [GJP18] is not trivial.

We also ignored blame assignment, an important component of contracts from a usability perspective. With a message passing model with potentially many forwardings, it seems subjective who to blame when some message is wrong, but either way, unlike with ordinary contracts where blame assignment can be implemented as a transformation on function arguments, with sesion types, it must be implemented as a transformation on the protocol itself.

Another important direction we can move towards is to extend the recurring theme of suppressing computation between some debug build and a production build, known as ghost code or ghosting [FGP16]. We observed that monitors seem more limited than ordinary contracts; we conjecture that an implementation and extension of ghosting to session types can fill the seeming gap that we noticed. In particular, we believe that blame assignment, which is fairly trivial on ordinary contracts, can be implemented for monitors using ghost code based on a preliminary investigation.

References

[Arn10] Rob Arnold. “C_0, an Imperative Programming Language for Novice Computer Scientists”. Available as Technical Report CMU-CS-10-145. M.S. Thesis. Department of Computer Science, Carnegie Mellon University, Dec. 2010.

[BP17] Stephanie Balzer and Frank Pfenning. “Manifest Sharing with Session Types”. In: *International Conference on Functional Programming (ICFP).* Extended version available as Technical Report CMU-CS-17-106R, June 2017. ACM, Sept. 2017, 37:1–37:29.

[CP10] Luís Caires and Frank Pfenning. “Session Types as Intuitionistic Linear Propositions”. In: *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010).* Paris, France: Springer LNCS 6269, Aug. 2010, pp. 222–236.

[FGP16] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. “The Spirit of Ghost Code”. In: *Formal Methods in System Design* 48.3 (2016), pp. 152–174. ISSN: 1572-8102. DOI: 10.1007/s10703-016-0243-x.

[GJP18] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. “Session-Typed Concurrent Contracts”. In: *Programming Languages and Systems.* Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 771–798. ISBN: 978-3-319-89884-1.

[Gom19a] Hannah Gommerstadt. personal communication. May 16, 2019.

[Gom19b] Hannah Gommerstadt. “Session-Typed Concurrent Contracts”. Available as Technical Report CMU-CS-19-119. PhD thesis. Carnegie Mellon University, Sept. 2019.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. “Linear Type Theory for Asynchronous Session Types”. In: *Journal of Functional Programming* 20.1 (Jan. 2010), pp. 19–50.

[Hon93] Kohei Honda. “Types for Dyadic Interaction”. In: *4th International Conference on Concurrency Theory.* CONCUR’93. Springer LNCS 715, 1993, pp. 509–523.

[HPC96] Joshua S. Hodas, Frank Pfenning, and Iliano Cervesato. “Efficient Resource Management for Linear Logic Proof Search”. In: *International Workshop on Extensions of Logic Programming — ELP’96.* Ed. by R. Dyckhoff, H. Herre, and P. Schröder-Heister. Leipzig, Germany: Springer-Verlag LNAI 1050, Mar. 1996, pp. 67–81.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. “Language Primitives and Type Discipline for Structured Communication-Based Programming”. In: *7th European Symposium on Programming Languages and Systems (ESOP 1998).* Springer LNCS 1381, 1998, pp. 122–138.

[MM84] Don P. Mitchell and Michael Merritt. “A Distributed Algorithm for Deadlock Detection and Resolution”. In: *Symposium on Principles of Distributed Computation (PODC 1984).* ACM. Vancouver, British Columbia, Aug. 1984, pp. 282–284.

[Pfe10] Frank Pfenning. *C0: Specification and Verification in Introductory Computer Science*. `http://c0.typesafety.net`. 2010.

[Pfe18] Frank Pfenning. *C0 Reference*. `http://c0.typesafety.net/doc/c0-reference.pdf`. Mar. 2018.

[Pfe19] Frank Pfenning. personal communication. Sept. 13, 2019.

[Pfe95] Frank Pfenning. "Refinement Types". In: *Proceedings of the Workshop on Logic, Domains and Programming Languages*. Ed. by K. Keimel et al. Invited talk. Darmstadt, Germany: Technical University Darmstadt, 1995.

[Rey83] John C. Reynolds. "Types, Abstraction, and Parametric Polymorphism". In: *Information Processing 83*. Ed. by R.E.A. Mason. Elsevier, Sept. 1983, pp. 513–523.

[WPP17] Max Willsey, Rokhini Prabhu, and Frank Pfenning. "Design and Implementation of Concurrent C0". In: *Electronic Proceedings in Theoretical Computer Science* 238 (Jan. 2017), pp. 73–82. DOI: `10.4204/EPTCS.238.8`.

A Shared Integer with Monotonicity Monitor Code

Listing 16: Full code showcasing the monitor for a shared integer with its implementation.

```
typedef <?choice shd_prov> A;
typedef <#; A> S;
choice shd_prov {
    <?int; S> Set;
    <!int; S> Get;
};

// checks if Set messages are monotonically increasing
S #c mon(S #d, int prev)
{
    A $c = (A)#c; // accept
    A $d = (A)#d; // acquire
    switch($c)
    {
        case Set: {
            int n = recv($c);
            assert(n >= prev);
            prev = n;
            $d.Set;
            send($d, n);
            break;
        }
        case Get: {
            $d.Get;
            int n = recv($d);
            send($c, n);
            break;
        }
    }
    #d = (S)$d; // release
    #c = (S)$c; // detach
    #c = mon(#d, prev);
}

S #c shd_int()
//@monitor #c = mon(#c, 0x80000000);
{
    int n = 0; // default
    while (true)
    {
        A $c = (A)#c; // accept
        switch($c)
        {
            case Set: {
                n = recv($c);
                break;
            }
            case Get: {
                send($c, n);
                break;
            }
        }
        #c = (S)$c; // detach
    }
}
```

B Binary Search Tree with Monitor Code

The code is based off an implementation in the programming language SILL by Gommerstadt [Gom19a].

Listing 17: Full code showcasing the split join based binary search tree.

```
typedef <?choice bst> bst;
choice bst {
    <?int; !bst; !bool; !bst; > split;
    <?bst; bst>                 join;
    <?int; ?int; bst>           chk;
};

bst $c mon(bst $d, int lo, int hi)
{
    switch($c)
    {
    case split:
    {
        int k = recv($c);
        $d.split;
        send($d, k);
        bst $l2 = recv($d);
        bool b = recv($d);
        bst $r2 = recv($d);
        wait($d);
        assert(!b || (lo < k && k < hi));
        send($c, $l2);
        send($c, b);
        send($c, $r2);
        close($c);
    }
    case join:
    {
        bst $t = recv($c);
        //@monitor $t = mon($t, hi, 0x7FFFFFFF);
        $d.join;
        send($d, $t);
        $c = mon($d, lo, 0x7FFFFFFF);
    }
    case chk:
    {
        $d.chk;
        int l = recv($c);
        int u = recv($c);
        send($d, l);
        send($d, u);
        $c = mon($d, lo, hi);
    }
    }
}

bst $t node(bst $l, bst $r, int k)
{
    switch($t)
    {
    case split:
    {
        int pivot = recv($t);
        if (pivot < k)
        {
            $l.split;
            send($l, pivot);
            bst $l2 = recv($l);
            bool b = recv($l);
            bst $r2 = recv($l);
            wait($l);
            send($t, $l2);
            send($t, b);
            bst $r3 = node($r2, $r, k);
            //@monitor $r3 = mon($r3, pivot, 0x7FFFFFFF);
            send($t, $r3);
            close($t);
        }
        else if (pivot == k)
        {
            send($t, $l);
            send($t, true);
            send($t, $r);
            close($t);
        }
        else // pivot > k
        {
            $r.split;
            send($r, pivot);
            bst $l2 = recv($r);
            bool b = recv($r);
            bst $r2 = recv($r);
            wait($r);
            bst $l3 = node($l, $l2, k);
            //@monitor $l3 = mon($l3, 0x80000000, pivot);
            send($t, $l3);
            send($t, b);
            send($t, $r2);
            close($t);
        }
    }
    case join:
    {
        bst $t2 = recv($t);
        $r.join;
        send($r, $t2);
        $t = node($l, $r, k);
    }
    case chk:
    {
        int lower = recv($t);
        int upper = recv($t);
        assert(lower <= k && k <= upper);
        $l.chk; // left side’s new bound: [lower, k]
        send($l, lower);
        send($l, k);
        $r.chk; // right side’s new bound: [k, upper]
        send($r, k);
        send($r, upper);
        $t = node($l, $r, k);
    }
    }
}

bst $t leaf()
{
    switch($t)
    {
    case split:
    {
        int k = recv($t);
        bst $l = leaf();
        bst $r = leaf();
        send($t, $l);
        send($t, false);
        send($t, $r);
        close($t);
    }
    case join:
    {
        bst $t2 = recv($t);
        $t = $t2;
    }
    case chk: // trivially a valid bst
    {
        int u = recv($t);
        int r = recv($t);
        $t = leaf();
    }
    }
}

bst $t singleton(int a)
{
    bst $l = leaf();
    bst $r = leaf();
    $t = node($l, $r, a);
```

```
}

bst $t insert(bst $t1, int a)
{
    bst $m = singleton(a);
    $t1.split;
    send($t1, a);
    bst $l = recv($t1);
    bool b = recv($t1);
    bst $r = recv($t1);
    wait($t1);
    $m.join;
    send($m, $r);
    $l.join;
    send($l, $m);
    $t = $l;
}

bst $t delete(bst $t1, int a)
{
    $t1.split;
    send($t1, a);
    bst $l = recv($t1);
    bool b = recv($t1);
    bst $r = recv($t1);
    wait($t1);
    $l.join;
    send($l, $r);
    $t = $l;
}

// send bool then become bst
<!bool; bst> $res find(bst $t, int a)
{
    $t.split;
    send($t, a);
    bst $l = recv($t);
    bool b = recv($t);
    bst $r = recv($t);
    wait($t);

    send($res, b);
    if (b)
    {
        $res = node($l, $r, a);
    }
    else
    {
        $l.join;
        send($l, $r);
        $res = $l;
    }
}
```

C Global Deadlock Detection Monitor Code

Listing 18: Full code showcasing the monitor for the global overseer.

```
typedef struct process_status process_status;
struct process_status {
    int pid;  // process id
    bool acquiring; // is it in the middle of acquiring?
    int acquiring_res; // if acquiring, should hold the resource id that is being acquired
    int_list* acquired_res; // list of acquired resources
};

typedef struct pstatus_list pstatus_list;
struct pstatus_list {
    process_status* ps;
    pstatus_list* next;
};

process_status* new_process_status(int p);

// true if list contains r
// should be called when will_release is received
bool found_res(int_list* acquired_res, int r);

// removes the first occurance of x from int_list
int_list* remove_res(int_list* acquired_res, int x);

// returns pointer to a process status corresponding to p
// if it cannot find it, returns a null pointer
process_status* find(pstatus_list* pss, int p);

// should initially be attached with a null pointer
soverseer #sa verify_protocol(soverseer #sb, pstatus_list* pss)
{
    loverseer $la = (loverseer)#sa; //acc
    loverseer $lb = (loverseer)#sb; //acq
    switch ($la) {
        case tryacq: {
            int p = recv($la);
            int r = recv($la);
            process_status* ps = find(pss, p);
            if (ps == NULL)
            {
                // add new ps
                ps = new_process_status(p);
                pstatus_list* new_pss = alloc(struct pstatus_list);
                new_pss->ps = ps;
                new_pss->next = pss;
                pss = new_pss;
            }
            assert(!ps->acquiring); // should not be in the middle of acquiring something
            ps->acquiring = true;
            ps->acquiring_res = r;
            $lb.tryacq;
            send($lb, p);
            send($lb, r);
            break;
        }
        case didacq: {
            int p = recv($la);
            int r = recv($la);
            process_status* ps = find(pss, p);
            assert(ps != NULL); // ps should exist since tryacq must have been called
            assert(ps->acquiring);
            assert(ps->acquiring_res == r);
            int_list* new_acq_res = alloc(struct int_list);
            new_acq_res->n = r;
            new_acq_res->next = ps->acquired_res;
            ps->acquired_res = new_acq_res;
            ps->acquiring = false;
            $lb.didacq;
            send($lb, p);
            send($lb, r);
            break;
        }
        case willrel: {
            int p = recv($la);
            int r = recv($la);
            // should only release acquired resources, also should not be currently acquiring
            process_status *ps = find(pss, p);
            assert(ps != NULL);
            assert(!ps->acquiring);
            assert(found_res(ps->acquired_res, r));
            int_list* new_acq_res = remove_res(ps->acquired_res, r);
            ps->acquired_res = new_acq_res;
            $lb.willrel;
            send($lb, p);
            send($lb, r);
            break;
        }
    }
    #sb = (soverseer)$lb;
    #sa = (soverseer)$la;
    #sa = verify_protocol(#sb, pss);
}
```