On Session Typed Contracts for Imperative Languages

Chuta Sano (andrew id: csano)

November 15, 2019

1 Abstract

We formalize an imperative programming language with session typed channels and adapt results by Gommerstadt et al [GJP18] of linear session typed contracts, or monitors, in a functional setting to the new imperative setting. We further postulate the semantics for corresponding monitors for shared session types. Finally, we introduce extensive case studies of linear and shared monitors in CC0, a C like language with session types, which we augment with monitors.

2 Introduction

C0 is a C-like language used in CMU's 15-122 to teach imperative programming, providing type and memory safety. Furthermore, C0 supports contracts, which help to enforce a disciplined style of programming [Pfe; Arn10]. Contracts are essentially runtime checks with no observable effect except on contract violation; a program without any contract violations will be indistinguishable to an equivalent program without the contracts. Since contracts help catch unintended program states, it is not only useful for a teaching environment but also for industry, where code is updated frequently on large projects.

Contracts in C0 comes 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 value or invariants of parameters that are passed by reference. Loop invariants are merely assertions that execute in the beginning of a loop. See Listing 1 for an example of the four contracts.

Concurrent C0, or CC0, was later developed which added inter-process communication over linear session-typed channels to C0 [WPP17]. Shared session-types were then implemented over CC0 [BP17]. A process providing a channel can be implemented in a similar style to an ordinary function; see Listing 2 for an example of a process implemented in CC0.

Listing 1: Syntax and usage of contracts in C0

```
int sqrt(int x)
//@requires x > 0;
//@ensures \result > 0 && \result <= x;
{
    ...
    //@assert some_bool_expression;
    ...
    while (condition)
    //@loop_invariant some_bool_expression_2;
    {
        ...
    }
    ...
}
```

Listing 2: Parallel Fibonacci in CC0

```
typedef <!int;> lint;

lint $c fib(int n)
{
    if (n == 0 || n == 1) {
        send($c, 1);
    }
    else {
        lint $f1 = fib(n-1);
        lint $f2 = fib(n-2);
        int n1 = recv($f1);
        int n2 = recv($f2);
        wait($f1);
        wait($f2);
        send($c, n1+n2);
    }
    close($c);
}

int main()
{
    lint $f5 = fib(5);
    int f5 = recv($f5);
    wait($f5);
    //@assert f5 == 8;
    return 0;
}
```

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 trivially translate to the imperative setting of CC0. Furthermore, there is no known work of a corresponding system of monitors for shared session types.

Thus, we have two main theoretical goals: the adaptation of linear monitors from functional to imperative and the proposal of shared monitors (in an imperative setting) that correspond with the known semantics of linear monitors.

Section 3 introduces A, which is a minimal imperative language with session types, and its corresponding type system. Section 4 introduces linear monitors and the corresponding additions to the type system. Section 5 introduces shared monitors. Section 6 contains various case studies and examples showcasing both variants of monitors in CC0.

3 Type system for A

3.a Abstract Syntax

Sort	Abstract Form	Remarks
Metavariable	c, d	Linear channel variable
	h	Shared channel variable
	x	Ordinary variable
	a	Generic (ordinary, linear, shared) variable
	p	Linear process name
	k	Shared process name
	f	Function name
	l	Choice names (labels)
Program pg	$dcl\ pg$	List of declarations
Decl dcl	$tp\ f(\overline{tp\ x}, \overline{stp\ d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}]\{s\}$	Function with contracts
	$stp\ c\ p(\overline{tp\ x}, \overline{stp\ d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}]\{s\}$	Linear process with contracts
	$shtp\ g\ k(\overline{tp\ x}, \overline{stp\ d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}]\{s\}$	Shared process with contracts
Stmt s	$\mathbf{return}\ e$	Return small type
	$tp\ x$	Declare fresh ordinary variable
	$x = e$	Ordinary variable assignment
	$x = f(\overline{e}, \overline{d})$	Channel dependent function call
	$d' = d$	Forward
	$d' = p(\overline{e}, \overline{d'})$	Spawn linear process
	$a = \mathbf{recv}(d)$	Receive from channel d
	$d.l$	Send choice label
	$\mathbf{send}(d, e)$	Send small type
	$\mathbf{send}(d, a)$	Send channel or shannel
	$\mathbf{wait}(d)$	Wait for channel to close
	$\mathbf{close}(c)$	Close channel
	$d = \mathbf{acq}(h)$	Acquire (downshift)
	$h = \mathbf{rel}(d)$	Release (upshift)
	$d = \mathbf{acc}(h)$	Accept (downshift)
	$h = \mathbf{det}(d)$	Detach (upshift)
	$s; s$	Sequence
	$\mathbf{if}(e)\ \{s\}\ \{s\}$	If else statement
	$\mathbf{while}(e)\ \{s\}$	While statement
	$\mathbf{switch}(d)\ \{\overline{l : s}\}$	Pattern matching over choices
	$\mathbf{assert}(e)$	Assert (contract)

Sort	Abstract Form	Remarks
Expression e	$\mathbf{true}, \mathbf{false}$	Boolean constants
	$f(\overline{e})$	Function call (no channels)
	x	Ordinary variable
Persistent type tp	otp	Ordinary Type
	$shtp$	Shared session type
Ordinary Type otp	$bool$	Boolean
Signature type $sigtp$	$(\overline{tp}, \overline{stp}) \to tp$	Function type
	$(\overline{tp}, \overline{stp}) \to stp$	Linear process type
	$(\overline{tp}, \overline{stp}) \to shtp$	Shared process type
Directional type dtp	$!tp$	Provider sends
	$?tp$	Provider receives
	$!\mathbf{choice}\{\overline{l, stp}\}$	Internal choice
	$?\mathbf{choice}\{\overline{l, stp}\}$	External choice
	$\#$	Shift point
	1	Unit
Shared session type $shtp$	$[\#; \overline{dtp}]$	
Linear session type stp	$\{\overline{dtp}\}$	

3.b Remark

This language is a slightly modified subset of CC0. As a subset, practical primitive types such as **int** and **double** and type constructors such as arrays, pointers, and structs are ommited for simplicity. Similarly, expressions and/or statements that are effectively sugaring, such as for loops, are also ommited.

The slight modification in this language is that we are forbidden from referencing linear session variables at expression level. In particular, $\mathbf{recv}(-)$ 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 recvs and arbitrary functions, which are already undefined behaviors in CC0 due to a lack of specified evaluation order of function arguments. We also group shared session types with persistent types instead of with linear session types since shared session types more closely resemble ordinary types than linear session types in most cases.

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.

3.c Type Judgements

3.c.1 Notation/Assumptions

Abstract Object	Common variable(s)
Persistent variable (small and shared)	x
Providing linear channel	c
Using linear channel	d
Providing shared channel	g
Using shared channel	h
Generic providing channel (linear or shared)	z
Function	f
Linear process	p
Shared process	k
Persistent type	τ
Linear session type	A
Shared session type	κ
Linear or shared session type	Z
Directional type	ψ

Lists Lists in the form $\overline{x}$ are assumed to be grouped right associative; that is,

$$\overline{s} = (s_1(s_2(\cdots)))$$

This is mostly relevant in resolving any parsing ambiguities in a sequence of statements.

Variable names As usual, variable names, including persistent, channels, functions, linear processes, and shared processes are all assumed to be 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.

3.c.2 Program

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

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

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

where declaration judgements are introduced in section 3.c.3.

3.c.3 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. Since $\textbf{req}(e)$ is a pre-condition, it is checked as an $\textbf{assert}(e)$. For $\textbf{ens}(e)$, we similarly check it as an $\textbf{assert}(e)$ 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 4.c.5 which justifies this decision.

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

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

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

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

where statement judgements are introduced in section 3.c.4 and expression judgements are introduced in section 3.c.5.

3.c.4 Statements

To typecheck statements, we maintain a context consisting of the persistent types:

$$\Gamma \triangleq \cdot \parallel \Gamma, a : tp$$

and a linear context Δ:

$$\Delta \triangleq \cdot \parallel \Delta, d : A$$

Δ structurally is Γ for linear channels but operates very differently; we use a two-state formulation since statements can progress linear channels: $\Delta \setminus \Delta'$. Γ, on the other hand, does not require a two-state formulation and therefore operates in the usual way.

Next, we introduce two meta-variables $\Delta_\downarrow$ and $A_\uparrow$:

$$\Delta_\downarrow \triangleq \Delta \parallel \downarrow$$
$$A_\uparrow \triangleq A \parallel \uparrow$$

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:

$$\Gamma\ ;\ \Delta \setminus \Delta'_\downarrow \vdash s :: \tau$$
$$\Gamma\ ;\ \Delta \setminus \Delta'_\downarrow \vdash s :: (z : A \setminus z' : A'_\uparrow)$$

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

$$\Theta \triangleq \tau \parallel (c : A \ \backslash \ c : A'_{\uparrow})$$

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

$$\Gamma \ ; \ \Delta \ \backslash \ \Delta'_{\downarrow} \vdash s :: \Theta$$

Statement Judgement Rules

We first introduce the two presumption 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 Δ', A'.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma\ \vdash e :: \mathbf{bool} \qquad \Gamma\ ;\ \Delta\ \setminus\ \Delta_{\downarrow}\ \vdash s :: (z : Z\ \setminus\ z' : Z_{\uparrow})}{\Gamma\ ;\ \Delta\ \setminus\ \Delta\ \vdash \mathbf{while}(e)\ \{s\} :: (z : Z\ \setminus\ z : Z)}\ \text{(STM-WHILEB)}$$

$$\frac{}{\Gamma\ ;\ d : A \setminus\ \downarrow \vdash c = d :: (c : A\ \setminus\ c :\uparrow)}\ \text{(STM-FWD1)}$$

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

$$\frac{}{\Gamma\ ;\ \Delta, d' : A\ \setminus\ \Delta, d : A\ \vdash d = d' :: \Theta}\ \text{(STM-FWD2)}$$

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

$$\frac{}{\Gamma, h : [\#; \overline{\psi}]\ ;\ \Delta\ \setminus\ \Delta, d : \{\overline{\psi}\}\ \vdash d = \mathbf{acq}(h) :: \Theta}\ \text{(STM-ACQ)}$$

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

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

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

3.c.5 Expressions

As a result of an imperative style language with its primary focus on linear session types, which are typed at statement level, expression checking is fairly lackluster. 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.

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

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

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

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

3.d Example

3.d.1 Remark

Although the formal rules will be written in terms of A, many non-trivial examples of code will be introduced in CC0 for aesthetic reasons. Here, we point out some stylistic and syntactic differences between A and CC0. In general, as per keeping the specification of A 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 A	Notes
$\$d, \$c$	d, c	Identifiers of linear channels in CC0 are preceded by an ampersand
$\#g, \#h$	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
$\tau x = \dots$	$\tau x; x = \dots$	CC0 allows an assignment on declaration
typedef τn	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.

3.d.2 Parallel Fibonacci

Listing 3: Parallel Fibonacci in CC0

```
1  <!int;> $c fib(int n)
2  {
3      if (n == 0 || n == 1) {
4          send($c, 1);
5          close($c);
6      } else {
7          <!int;> $f1 = fib(n-1);
8          <!int;> $f2 = fib(n-2);
9          int x1 = recv($f1); wait($f1);
10         int x2 = recv($f2); wait($f2);
11         send($c, x1 + x2);
12         close($c);
13     }
14 }
```

We will briefly walk through the typechecker for this relatively simple linear process.

At line 2, Γ solely consists of x 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 11, 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 11 and 12, 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 presumption 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.

4 Linear Monitors and Partial Identity

4.a 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.

Gommerstadt et al formulates session typed contracts as intermediary processes (monitor) that attach to a channel and listen along the channel [GJP18]. See diagram 1 for a visual representation of the monitor.

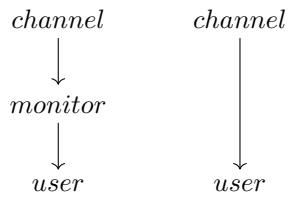

diagram 1: Left: Set up with monitor. Right: Set up without. The monitor acts as an eavesdropper between the user and the channel.

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.

4.b Language

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

$$
\begin{aligned}
\mathit{decl} \triangleq\ & \mathit{tp}\ f(\overline{\mathit{tp}\,x}, \overline{\mathit{stp}\,d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\} \\
& |\mathit{stp}\ c\ p(\overline{\mathit{tp}\,x}, \overline{\mathit{stp}\,d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\} \\
& |\mathit{shtp}\ g\ k(\overline{\mathit{tp}\,x}, \overline{\mathit{stp}\,d})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\} \\
& |\mathbf{mon}(\mathit{stp}\ c\ p(\mathit{stp}\,d, \overline{\mathit{otp}\,x})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\}) \\
\mathit{stmt} \triangleq\ & \ldots \\
& |\mathbf{mon}(d = p(d, \overline{e})) \\
\mathit{sigtp} \triangleq\ & \ldots \\
& |\mathbf{mon}((\mathit{stp}, \overline{\mathit{tp}}) \to \mathit{stp})
\end{aligned}
$$

All declarations now support monitors over their parameters and providing channel if appropriate. Note that unlike ordinary contracts, there is no syntactic distinction between monitors corresponding to requires and monitors corresponding to ensures because it can be inferred. This is discussed in section 4.c.5. 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 types (shared channels are not allowed). We postulate that the addition of shared channels as parameters in monitors requires a generalization of ghost types to session types. We briefly discuss this generalization in section TODO. 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 $\mathbf{mon}(d =$

$p(d, \dots))$ to $d' = p(d, \dots); d = d'$ in an implementation with debugging enabled to effectively enable the monitor.

Finally, a valid linear monitor's typing signature will explicitly be marked as so for simplification. In practice, a global table with a listing of all valid linear monitors will serve the same purpose.

4.c Type Judgements

4.c.1 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 4.c.5. Therefore, we will assume that declarations do not have any contracts, as in section 3.c.3.

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

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.

4.c.2 Statements

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

4.c.3 Partial Identity

Intuitively, partial identity is checked by maintaining a queue of messages to be sent by either the providing channel c or the client d depending on the direction.

We define a member of the queue as follows:

$$q \triangleq \mathbf{Pers}(x) | \mathbf{Ch}(d) | \mathbf{Label}(l) | \mathbf{End}$$

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

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

$$\omega \triangleq (- \Leftrightarrow -)$$

with ω as the metavariable for arbitrary tracking contexts. The two horizontal arrows $(\Leftarrow, \Rightarrow)$ denote the direction of the message with the metavariable $\Leftrightarrow$ taking place of either of them. Since $\Leftarrow$ asserts that the message is flowing from client to providing channel, this is alternatively interpreted as the queue being associated with the providing channel, and vice versa.

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:

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

where Γ and Δ are analogous to the ones in statement judgements, Σ is the function and process definition environment, and $\overline{q}$ is the message queue. Note that the second tracking context allows a different tracking client d'.

The first rule is a presumption rule similar to the statement judgements. In this case, a proper termination allows us to infer any arbitrary state. As in the statement judgements, these will be encountered in control flow statements like if or while.

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

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.

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

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

$$\frac{}{\Gamma, h : [\#; \overline{\psi}]; \Delta \setminus \Delta, d : \{\overline{\psi}\}; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash \Theta \ \mathbf{pid}} \text{ (PID-ACQ)}$$

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

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

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

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

$$\frac{\mathbf{Ch}(d') \notin \overline{q}}{\Gamma; \Delta, d' : A \setminus \Delta, d : A; (\omega \setminus \omega); (\overline{q} \setminus \overline{q}) \vdash d = d' \ \mathbf{pid}} \text{ (PID-FWD1)}$$

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

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

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

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

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

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

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

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

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

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

The rules so far are for the most part identical to the statement judgements since the tracking context and the queue were not used. The frequent $\notin$ clauses ensure that variables in the queue do not get 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.

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

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

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

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

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

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.

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

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

recv and **send** record and consume $\mathbf{Pers}(x)$ and $\mathbf{Ch}(d')$ messages in the queue respectively.

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

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

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

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

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

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

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

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

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

Switch statements and sending labels again follow a similar pattern:

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

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

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

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

4.c.4 Forwarding and Spawning

Forwarding the client simply changes the client in the tracking context, and forwarding the provider (with the client) requires the linear context and queue to be empty.

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((c : A \Leftrightarrow d : A') \setminus (c : A \Leftrightarrow d' : A')); (\overline{q} \setminus \overline{q}) \vdash d' = d\ \mathbf{pid}}\ \text{(PID-FWD2)}$$

$$\frac{}{\Gamma; \cdot \setminus \cdot; ((c : A \Leftrightarrow d : A) \setminus (\uparrow\Leftrightarrow\downarrow)); (\cdot \setminus \cdot) \vdash c = d\ \mathbf{pid}}\ \text{(PID-FWD3)}$$

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

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

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). In section TODO, we provide a detailed discussion of how arbitrary spawning, which generalizes various aspects of the partial identity checks, can be implemented.

4.c.5 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:

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

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.

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

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.

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

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.

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

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.

4.d Examples

Listing 4: 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);
```

```
15      close($c);
16  }
```

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 **End** is now the only queue member. Therefore, is_pos2 is also a valid monitor.

Listing 5: List Monitor in CC0

```
1  choice list {
2      <> Nil;
3      <!int; !choice list> Cons;
4  };
5  typedef <!choice list> list;
6
7  list $c nil() {
8      $c.Nil;
9      close($c);
10 }
11
12 list $c cons(int n, list $d) {
13     $c.Cons;
14     send($c, n);
15     $c = $d;
16 }
17
18 list $c allbigger_than(list $d, int threshold) {
19     switch ($d) {
20     case Nil:
21         wait($d);
22         $c.Nil;
23         close($c);
```

```
24      case Cons: {
25          int res  = recv($d);
26          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 $\mathbf{Label}(Nil)$. 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, $\overline{q} = \mathbf{Label}(Cons), \mathbf{Pers}(res)$. In line 27, we send the label Cons, which is allowed because the head of the queue is $\mathbf{Label}(Cons)$. 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.

5 Shared Monitors

We next introduce shared monitors. As it turns out, partial identity for shared monitors is fairly intuitive. 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.

$$
\begin{aligned}
&\mathit{decl} \triangleq \ldots \\
&\quad |\mathbf{mon}(\mathit{shtp}\ g\ k(\mathit{shtph}, \overline{\mathit{tp}\,x})[\overline{\mathbf{req}(e)}, \overline{\mathbf{ens}(e)}, \overline{\mathbf{mon}(d = p(d, \overline{e}))}]\{s\}) \\
&\mathit{stmt} \triangleq \ldots \\
&\quad |\mathbf{mon}(g = p(h, \overline{e})) \\
&\mathit{sigtp} \triangleq \ldots \\
&\quad |\mathbf{mon}((\mathit{shtp}, \overline{\mathit{tp}}) \to \mathit{shtp})
\end{aligned}
$$

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

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

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 – 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.

5.a Type Judgements

5.a.1 Declaration

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

5.a.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 is 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 receiving 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.

$$q \triangleq \ldots | \mathbf{Dshift} | \mathbf{Ushift}$$

We first introduce the two shift rules. Surprisingly, both rules anticipate a right direction since both accept and detach are interpreted as messages from the user. This can be explained by TODO.

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

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

A monitor involving shared channels consists of:

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

where 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 process and hold it, which can violate partial identity.

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((g : [\#\overline{\psi}] \Rightarrow h : [\#\overline{\psi}]) \setminus (c : \{\overline{\psi}\} \Rightarrow h : [\#\overline{\psi}])); (\cdot \setminus \mathbf{Ushift}) \vdash c = \mathbf{acc}(g) \ \mathbf{pid}} \text{ (PID-ACC2)}$$

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

$$\frac{}{\Gamma; \Delta \setminus \Delta; ((c : \{\overline{\#\psi}\} \Rightarrow d : \{\overline{\#\psi}\}) \setminus (g : [\overline{\#\psi}] \Rightarrow d : \{\overline{\#\psi}\})); (\cdot \setminus \mathbf{Dshift}) \vdash g = \det c \ \mathbf{pid}} \text{ (PID-DET2)}$$

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

Again, we do notice a surprising phenomenon that the direction of the messages do not seem balanced. This is explained by TODO.

5.b Acknowledgement of Accept/Acquire

Unfortunately, the model of accept, acquire, linear code, detach, and then release is not sufficient for partial identity. Consider a typical situation where a shared channel, provided by a process p, is monitored by a monitor mon and is used by some user u as in diagram 2.

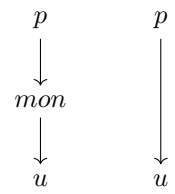

diagram 2: Left: Set up with monitor. Right: Set up without. By partial identity, the two cannot be differentiated aside from termination.

On the right hand side, when u acquires a channel provided by p, the statements proceding it are blocked until p actually accepts. However on the left hand side, when u acquires a channel provided by p, due to the intervention of mon, the statements proceding it are blocked only until mon accepts. In particular, this means that partial identity can only be achieved if mon acquires from p simultaneously, which is not possible.

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**. In general, the session types

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

should respectively transform to

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

Then, outside a monitor, a statement of form

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

should transform to

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

and similarly, a statement of form

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

should transform to

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

where x is fresh and therefore not referenced in s.

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 computation that does not affect the tracking context nor the queue. These sequences of statements should be transformed to

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

TODO proof of well typed -> still well typed?

5.c Examples

TODO after discussion

6 Case Studies

6.a Linear Monitors in CC0

6.b Shared Monitors in CC0

7 Conclusion

TODO

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.

[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.

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

[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.

8 Appendix

TODO

- cut2 generalizations on partial id
- ...