Sistemas de Tipos para Programação Concorrente Um Estudo de Caso*

Cláudio Henrique Oliveira Ribeiro[1], **Bruno Silvestre**[1], **Daniel Ventura**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia, GO – Brazil

claudio.henrique@egresso.ufg.br, {brunoos,ventura}@ufg.br

Abstract. *In this work, a case study of a Go program encoded as a process calculus term is presented, as a first step in proposing a Go session types system.*

Resumo. *O presente trabalho apresenta um estudo de caso da representação de um programa em Go em um cálculo de processos, como uma primeira abordagem para a proposta de um sistema de tipos de sessão para a linguagem.*

1. Introdução

Go é uma linguagem de programação estaticamente tipada, de fácil escrita e desenvolvida com foco em desempenho. Devido ao equilíbrio entre segurança e expressividade, obtido pela utilização dos tipos, e o melhor desempenho em relação as linguagens dinâmicas, ela vem sendo utilizada para substituir linguagens não tipadas [Donovan and Kernighan 2015]. O suporte a concorrência é provido na linguagem tanto através do compartilhamento de memória quanto através da troca de mensagens, sendo o último incentivado pela simplicidade em sua utilização. Apesar do sistema de tipos, verificações tais como se o tipo de valores enviados e recebidos coincidem, propriedades como não-interferência em canais de comunicação não são verificadas.

O cálculo de processos introduzido em [Vasconcelos 2012] apresenta uma noção de linearidade baseada no conceito de co-variáveis. Ao contrário da definição usual de comunicação, onde uma variável x representa o nome de um canal de comunicação utilizado "nas duas pontas" dessa comunicação, o cálculo utiliza co-variáveis representando as duas extremidades de um mesmo canal. Assim, ao invés de $\overline{x}v.P \mid x(z).Q \rightarrow P \mid Q[v/z]$ temos $(\nu xy)(\overline{x}v.P \mid y(z).Q) \rightarrow (\nu xy)(P \mid Q[v/z])$, para x e y co-variáveis. Dessa forma, a comunicação só ocorre como (nomes de) canais ligados por ν. Um canal de comunicação é então dito linear se cada uma de suas extremidades ocorre em apenas um processo, *i.e.* não ocorre em processos executando em paralelo. O sistema de tipos de sessão garante que, para um canal linear, o tipo do canal descreve exatamente a sua utilização durante uma sessão de comunicação. Ao contrário de outros tipos de sessão "lineares", termos tipados nesse cálculo não estão livres de *deadlocks* [Kobayashi 2002]. Uma especificação em Maude da versão algorítmica do sistema de tipos foi apresentada em [Restrepo et al. 2023] onde, além das propriedades garantidas pelo tipos, a verificação de *deadlocks* e/ou *locks* é realizada utilizando um *model checker*.

O presente trabalho apresenta um estudo de caso da representação no cálculo de processos de um programa em Go, como uma primeira abordagem para a proposta de um sistema de tipos de sessão para a linguagem.

*O presente trabalho foi realizado com apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) - UNIVERSAL 423212/2021-4

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv P$$
$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad (\nu xy)\mathbf{0} \equiv \mathbf{0}$$
$$(\nu xy)(\nu wz)P \equiv (\nu wz)(\nu xy)P \qquad P \mid (\nu xy)Q \equiv (\nu xy)(P \mid Q), \text{ if } x, y \notin \mathtt{fv}(P)$$

[LinCom]	$(\nu xy)(\overline{x}v.P \mid \mathtt{lin}\ y(z).Q \mid R)$	$\rightarrow$	$(\nu xy)(P \mid Q[v/z] \mid R)$
[UnCom]	$(\nu xy)(\overline{x}v.P \mid \mathtt{un}\ y(z).Q \mid R)$	$\rightarrow$	$(\nu xy)(P \mid Q[v/z] \mid \mathtt{un}\ y(z).Q \mid R)$
[Case]	$(\nu xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid R)$	$\rightarrow$	$(\nu xy)(P \mid Q_j \mid R)$, if $j \in I$
[Par]	if $P \rightarrow P'$ then $P \mid Q$	$\rightarrow$	$P' \mid Q$
[Res]	if $P \rightarrow P'$ then $(\nu xy)P$	$\rightarrow$	$(\nu xy)P'$
[Struct]	if $P \equiv P', Q \equiv Q', P' \rightarrow Q'$ then P	$\rightarrow$	Q

Figura 1. Congruências e Semântica Operacional de $\mathtt{s}\pi$.

2. Preliminares

2.1. Cálculo de Processos e Tipos de Sessão

Apresentamos o cálculo de processos $\mathtt{s}\pi$ [Restrepo et al. 2023] omitindo os processos condicionais ($\mathtt{if}\ v\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$) e respectivas congruências, regras de redução e de tipos. A sintaxe dos **valores** v, **qualificadores** q e **processos** P em $\mathtt{s}\pi$ são definidas por:

$$v ::= x \mid \mathtt{true} \mid \mathtt{false} \qquad q ::= \mathtt{un} \mid \mathtt{lin}$$
$$P, Q ::= \mathbf{0} \mid \overline{x}v.P \mid q\ x(y).P \mid P_1 \mid P_2 \mid (\nu xy)P \mid x \triangleleft l.P \mid x \triangleright \{l_i : P_i\}_{i\in I}$$

onde $x \in \mathcal{X}$, um conjunto infinito de variáveis. A semântica operacional de $\mathtt{s}\pi$ é apresentada na Fig. 1 e apresentamos a seguir uma breve explicação sobre o significado dos construtores de processos. O processo $\mathbf{0}$ representa um processo que não executa nenhuma ação. No construtor de envio $\overline{x}v.P$ temos o envio de um valor v através do canal x e, após o envio, o processo dará continuidade como o processo P. Já o construtor de recebimento $q\ x(y).P$ é formado por um qualificador $q \in \{\mathtt{un}, \mathtt{lin}\}$, por um canal que ao receber um valor v dará continuidade como o processo P com as ocorrências livres de y substituídas por v, denotado por $P[v/y]$. $P_1 \mid P_2$ representa a execução paralela dos processos P_1 e P_2. O construtor de escopo (restrição de nomes) $(\nu xy)P$ relaciona as variáveis x e y, chamadas de co-variáveis, e restringe seu escopo ao processo P. O construtor $x \triangleright \{l_i : P_i\}_{i\in I}$ representa a oferta de um conjunto de processos P_i, cada um associado a um rótulo l_i. Por outro lado, o construtor de seleção $x \triangleleft l.P$ permite escolher um processo através do envio do rótulo l, continuando como o processo P. O conjunto $\mathtt{fv}(P)$ de variáveis livres é definido como esperado, onde $q\ x(y).P$ e $(\nu xy)P$ são ligantes, ou seja, $\mathtt{fv}(q\ x(y).P) = (\mathtt{fv}(P) \setminus \{y\}) \cup \{x\}$ e $\mathtt{fv}((\nu xy)P) = \mathtt{fv}(P) \setminus \{x, y\}$. Note que, na ação definida em [UnCom], $\mathtt{un}\ y(z).Q$ corresponde a um processo replicante, onde a comunicação é realizada através de uma cópia do processo, permitindo que essas ações continuem disponíveis para comunicações futuras (*e.g. ping server* [Kobayashi 2002]). Para simplificar a notação, omitiremos dos termos o qualificador $\mathtt{lin}$.

A sintaxe dos **pré-tipos** p e dos **tipos de sessão**, ou simplesmente tipos, T são definidas por:

$$p ::= ?T.T \mid !T.T \mid \&\{l_i : T_i\}_{i\in I} \mid \oplus\{l_i : T_i\}_{i\in I}$$
$$T, U ::= \mathtt{bool} \mid \mathtt{end} \mid q\ p \mid a \mid \mu a.T$$

onde $a \in \mathcal{T}$, conjunto infinito de variáveis de tipos. Os tipos recursivos $\mu a.T$ são considerados equi-recursivamente, ou seja, $\mu a.T \equiv T[\mu a.T/a]$. Um tipo T é **irrestrito**, denotado por $\mathtt{un}(T)$, se é $\mathtt{bool}$, $\mathtt{end}$ ou $\mathtt{un}\ p$, e chamados de **lineares**, denotado por $\mathtt{lin}(T)$, caso contrário. Dado um tipo T, seu **dual** $\overline{T}$ é definido por:

$$\overline{q\ ?T.U} = q\ !T.\overline{U} \qquad \overline{q\ \&\{l_i : T_i\}_{i\in I}} = q \oplus \{l_i : \overline{T_i}\}_{i\in I} \qquad \overline{\mathtt{end}} = \mathtt{end} \quad \overline{a} = a$$

$$\overline{q\ !T.U} = q\ ?T.\overline{U} \qquad \overline{q \oplus \{l_i : T_i\}_{i\in I}} = q\ \&\{l_i : \overline{T_i}\}_{i\in I} \qquad \overline{\mu a.T} = \mu a.\overline{T}$$

Contextos de tipo Γ são definidos por: $\Gamma ::= \emptyset \mid \Gamma, x : T$. Dizemos que $q(\Gamma)$ se $q(T)$ para todo $x : T \in \Gamma$. A **divisão** (*split*) de contextos é definida por:

$$\emptyset \circ \emptyset = \emptyset \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \mathtt{un}(T)}{\Gamma, x : T = (\Gamma_1, x : T) \circ (\Gamma_2, x : T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \mathtt{lin}\ p = (\Gamma_1, x : \mathtt{lin}\ p) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x : \mathtt{lin}\ p = \Gamma_1 \circ (\Gamma_2, x : \mathtt{lin}\ p)}$$

Intuitivamente, a divisão de contextos está bem definida quando variáveis com tipo linear são divididas entres os contextos, enquanto as variáveis designadas com tipos irrestritos são compartilhas. A **atualização** de contextos é definida por:

$$\frac{x : U \notin \Gamma}{\Gamma + x : T = \Gamma, x : T} \qquad \frac{\mathtt{un}(\Gamma)}{(\Gamma, x : T) + x : T = (\Gamma, x : T)}$$

Apresentamos a versão concisa do sistema [Vasconcelos 2012] (Fig. 2), enquanto a versão algorítmica, especificada em [Restrepo et al. 2023], é apresentada no Ap. A (Fig. 10). Para q_1, q_2 em [T-IN], se $q_1 = \mathtt{un}$ então $q_2 = \mathtt{un}$ enquanto se $q_1 = \mathtt{lin}$ então q_2 pode ser $\mathtt{lin}$ ou $\mathtt{un}$. Apresentamos uma discussão mais detalhada das características dos sistema de tipos ao apresentar os resultados do caso de estudo na Sec. 4.

2.2. Suporte à concorrência em Go

O suporte à concorrência em Go é realizado por meio das **go-rotinas** (*goroutines*), que permitem disparar novas linhas de execução, explorarando multiprocessadores, por exemplo. A comunicação entre as go-rotinas pode ser feita por meio de memória compartilhada ou canais. O uso de canais é incentivado pois facilita o entendimento sobre o fluxo de informações, sendo mais simples para programadores não acostumados com programação concorrente, do que ter que lidar com sincronização de estado (*locks*, semáforos, sinais, etc.), como é feito no caso de memória compartilhada.

Go-rotinas "Go-rotinas são funções que executam concorrentemente com outras go-rotinas em um mesmo espaço de endereçamento" (*Effective* Go, 2023, tradução nossa) [Go Language 2009]. Elas são conhecidas também como *threads* leves, pois elas são criadas e escalonadas em modo usuário. A *thread* do sistema operacional é considerada pesada pois sua criação ou escalonamento envolve a troca de contexto do modo usuário (da aplicação) para o modo *kernel* e depois retornando, consumindo muitos ciclos de CPU. Diversas go-rotinas são escalonadas em uma mesma *thread* do sistema operacional até que alguma go-rotina cause um bloqueio (por exemplo entrada/saída). Nesse caso uma nova *thread* é disparada para executar as demais go-rotinas, enquanto aquela que estava em execução agora está bloqueada.

$$\frac{\mathtt{un}(\Gamma)}{\Gamma \vdash \mathtt{true} : \mathtt{bool}}\ [\text{T-TRUE}] \qquad \frac{\mathtt{un}(\Gamma)}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}\ [\text{T-FALSE}] \qquad \frac{\mathtt{un}(\Gamma)}{\Gamma, x : T \vdash x : T}\ [\text{T-VAR}]$$

$$\frac{}{\Gamma \vdash \mathbf{0}}\ [\text{T-INACT}] \qquad \frac{\Gamma_1 \vdash P \qquad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q}\ [\text{T-PAR}] \qquad \frac{\Gamma_1, x : T, y : \overline{T} \vdash P}{\Gamma_1 \vdash (\nu xy)P}\ [\text{T-RES}]$$

$$\frac{\Gamma_1 \vdash x : q\,!T.U \qquad \Gamma_2 \vdash v : T \qquad \Gamma_3 + x : U \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash \overline{x}v.P}\ [\text{T-OUT}]$$

$$\frac{q_1(\Gamma_1 \circ \Gamma_2) \qquad \Gamma_1 \vdash x : q_2\,?T.U \qquad (\Gamma_2 + x : U), y : T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash q_1\,x(y).P}\ [\text{T-IN}]$$

$$\frac{\Gamma_1 \vdash x : q\,\&\{l_i : T_i\}_{i \in I} \qquad \Gamma_2 + x : T_i \vdash P_i \qquad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleright \{l_i : P_i\}_{i \in I}}\ [\text{T-BRANCH}]$$

$$\frac{\Gamma_1 \vdash x : q \oplus \{l_i : T_i\}_{i \in I} \qquad \Gamma_2 + x : T_j \vdash P \qquad j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \triangleleft l_j.P}\ [\text{T-SEL}]$$

Figura 2. Regras de Tipos para Valores e Processos

3. Caso de Estudo

Um *testbed* é um ambiente que possui facilidades para realizar testes, muitas vezes, focado em um domínio ou tecnologia. Por exemplo, *testbeds* com equipamentos 5G, outros com GPUs ou mesmo com grande poder computacional e armazenamento. É interessante que esses ambientes tenham acesso remoto e que possam ser compartilhados por diversos usuários (pesquisadores, por exemplo), permitindo que eles usufruam dessa infraestrutura que muitas vezes não está disponível em sua instituição.

O caso de estudo da formalização se refere a um controlador de um *testbed* de Internet das Coisas (IoT), onde o usuário pode ter acesso a diversos equipamentos como microcontroladores (Arduinos ou ESPs) equipados com diferentes tecnologias de comunicação (Zigbee ou LoRa), sensores e atuadores.

A Figura 3 ilustra a interação com o *testbed*. O usuário, via Internet, acessa os recursos do ambiente por meio de um servidor controlador, responsável por autenticação, reserva e agendamento de uso dos recursos e também segurança, impedindo acesso indevido ou que um usuário interfira no experimento de outro.

Para o funcionamento automatizado de um *testbed*, é necessária que uma infraestrutura de apoio seja construída. Por exemplo, para um ambiente simples oferecendo Arduinos como equipamentos, é necessário que o usuário possa remotamente enviar novos programas a serem implantados nesses equipamentos. Uma forma de fazer isso é ter, por exemplo, um microcomputador Raspberry Pi que irá receber o binário e realizar o processo de implantação (*flash*) do binário no Arduino.

Dessa forma, o usuário passa as informações para o controlador de qual Arduino deve ser programado, juntamente com o código binário para ser implantado, que então repassa internamente para o Raspberry Pi apropriado realizar a operação.

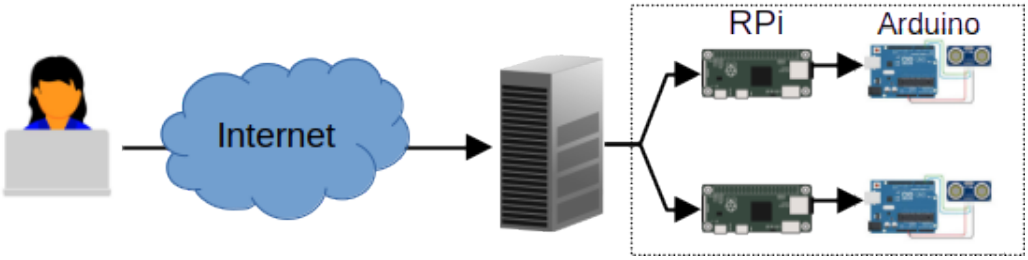

Figura 3. Exemplo de um *testbed* de IoT.

```
func client() {
  conn := net.DialTCP("tcp", nil, serverAddress)
  conn.Write(marshalRequest(request))

  data := make([]byte, dataSize)
  conn.Read(data)
  response := unmarshalResponse(data)
  conn.Close()
}
```

Figura 4. Pseudocódigo do cliente do *testbed*.

A Figura 4 apresenta uma simplificação de código em Go que o usuário executaria para se conectar com o servidor, enviar uma requisição com a identificação do Arduino e o código binário. Então, o cliente esperaria uma resposta do servidor indicando sucesso ou não da operação.

Na Figura 5 temos o código (simplificado) do servidor. Este espera por novas requisições de usuário (*listen.Accept*) e quando isso ocorre, ele recebe a requisição e dispara uma go-rotina *handleRequest* para processá-la, voltando (laço *for*) a esperar por novas requisições.

A go-rotina lê a requisição da rede, identifica qual o Raspberry Pi a ser acionado e repassa a requisição via rede a ele. A rotina então espera o resultado vindo do Raspberry Pi e então o repassa ao cliente.

Note que o servidor não espera o resultado do Raspberry Pi para atender uma nova requisição de um cliente. Dessa forma, várias requisições de programação de Arduinos podem ser tratadas de forma concorrente.

4. Resultados

Considerando as características da implementação do *testbed* descrito na seção anterior, podemos definir a representação do programa como um termo no cálculo de processos composto por três processos definidos em paralelo:

- Um cliente que se comunica com o servidor, enviando uma requisição contendo a identificação do Arduino que ele deseja utilizar e o código binário, o qual será implantado no Arduino.
- Um servidor em Go que recebe a requisição de um cliente, e dispara uma go-rotina para localizar o endereço do Raspberry Pi informado na requisição, abrir um canal de comunicação com ele e repassar o código binário.

```
func server() {
  listen := net.Listen("tcp", serverAddress)
  for {
    conn := listen.Accept()
    go handleRequest(conn)
 }
}

func handleRequest(client net.Conn) {
  data := make([]byte, dataSize)
  client.Read(data)
  request := unmarshalRequest(data)

  rpiAddr := rpiMap[request.RpiId]
  rpiConn := net.DialTCP("tcp", nil, rpiAddr)

  rpiConn.Write(request.File)
  rpiConn.Read(data)
  client.Write(data)

  client.Close()
  rpiConn.Close()
}
```

Figura 5. Pseudocódigo do servidor do *testbed*.

- Por fim, a go-rotina espera o resultado da ação do *Raspberry Pi* (sucesso ou erro), e o repassa para o cliente.

Consideramos que os clientes em comunicação com o servidor já tenham sido autenticados, e identificamos o *Raspberry Pi* com o seu *Arduino* associado, omitindo a comunicação entre esses componentes. Em relação a segurança citada na Seção 3, a restrição de acesso ao *Raspberry Pi* será obtida pela restrição de acesso ao seu canal de comunicação a apenas o servidor e a interferência no experimento de um usuário (cliente) evitada pela utilização de canais lineares de comunicação. Para representar a passagem do arquivo usaremos dados booleanos[1].

Nessa primeira proposta (Fig. 6), o cliente utiliza o canal compartilhado do servidor para iniciar sua conexão, enviando seu próprio canal privado (linear) de comunicação, satisfazendo os critérios de linearidade para o canal de comunicação cliente-servidor. Para permitir a escolha do *Raspberry Pi* pelo cliente, usamos os construtores de processo para oferta e escolha. Dessa forma, o servidor é capaz de estabelecer conexões com diferentes *Raspberry Pi*'s, de acordo à solicitação do cliente[2]. Com essa representação foi possível obter algumas informações sobre o tipo de sessão do processo, que detalhamos a seguir.

No processo do cliente temos que uma ação de seleção é executada pelo canal cs_1, seguida da ação de envio de um valor do tipo booleano pelo mesmo canal, continuando

[1] uma extensão de $s\pi$ para incluir outros tipos de dados, *e.g.* inteiros, pode ser obtida de forma direta.

[2] temos apenas uma opção de escolha, para incluir a etapa de escolha da análise.

$$P = (\nu cs_1 cs_1')(\nu ss')(\nu rr')(Cliente \mid Servidor \mid RPI)$$
$$Servidor = \mathtt{un}\ s(y).(y \triangleright \{Rpi : \mathtt{lin}\ y(file).\overline{r'} file.\mathtt{lin}\ r'(resp).\overline{y} resp.\mathbf{0}\})$$
$$Cliente = \overline{s'} cs_1'.cs_1 \triangleleft Rpi.\overline{cs_1}\mathtt{true}.\mathtt{lin}\ cs_1(x).\mathbf{0} \qquad RPI = \mathtt{un}\ r(file).\overline{r}\mathtt{true}.\mathbf{0}$$

Figura 6. Proposta 1

sua execução o canal espera receber um valor booleano e por fim encerrando sua execução juntamente ao uso do canal. Note que cada uso do canal deve ser qualificado com um q. Outra observação é que no tipo de sessão do $Cliente$ não está descrita a ação $\overline{s'} cs_1'$, isso se dá pelo fato que o canal s' é complementar a s que é um canal do processo do servidor. O tipo de s está descrito no tipo de sessão do servidor enquanto o tipo de s' é o seu dual (veja [T-RES] na Fig. 2):

$$Servidor : \mathtt{un}\ ?\mathtt{bool}.(\mathtt{lin}\ \&\{Rpi : \mathtt{lin}\ ?\mathtt{bool}.(\mathtt{un}\ !\mathtt{bool}.(\mathtt{lin}\ ?\mathtt{bool}.(\mathtt{lin}\ !\mathtt{bool}.\mathtt{end})))\})$$
$$Cliente : \mathtt{lin} \oplus (\mathtt{lin}\ !\mathtt{bool}.(\mathtt{lin}\ ?\mathtt{bool}.\mathtt{end})) \quad RPI : \mathtt{un}\ ?\mathtt{bool}.(\mathtt{lin}\ !\mathtt{bool}.\mathtt{end})$$

Baseado nessa análise dos tipos, foram identificadas incompatibilidades entre a definição do processo para o *Raspberry Pi* e as regras de uso de canais compartilhados. Para checar o RPI aplica-se a regra [T-IN], ou seja

$$\frac{\mathtt{un}(\Gamma_1 \circ \Gamma_2) \qquad \Gamma_1 \vdash r : \mathtt{un}\ ?T.U \qquad (\Gamma_2 + r : U), file : T \vdash \overline{r}\mathtt{true}.\mathbf{0}}{\Gamma_1 \circ \Gamma_2 \vdash \mathtt{un}\ r(file).\overline{r}\mathtt{true}.\mathbf{0}}$$

onde $T = \mathtt{bool}$ e $U = \mathtt{lin}\ !\mathtt{bool}.\mathtt{end}$. Assim, temos que $r : \mathtt{un}\ ?T.U \in \Gamma_1$ e, pela definição de $\Gamma_1 \circ \Gamma_2$ (Sec. 2), $r : \mathtt{un}\ ?T.U \in \Gamma_2$. Por outro lado, pela definição da atualização de contextos (Sec. 2) temos que $\Gamma_2 + r : U$ está bem definida só se U é irrestrito e $r : U \in \Gamma_2$, ou seja deveríamos ter que $\mathtt{un}\ ?T.U = U$. Intuitivamente, por se tratar de um canal compartilhado, o mesmo deve se manter ativo após seu uso, ou seja, tanto o processo original quanto a cópia utilizada na comunicação devem ser tipados.

Adequando a representação de forma semelhante a comunicação cliente-servidor, teríamos que cada *Raspberry Pi* possui um canal compartilhado com o servidor e, ao selecionar um dos *Raspberry Pi*'s, o servidor envia o canal de comunicação privada. Assim, a conexão de duas sessões distintas do servidor ao mesmo *Raspberry Pi* terá um canal exclusivo de comunicação. Contudo, devemos tomar uma cuidado extra no caso da comunicação servidor-microcomputador. Definindo o processo $P = (\nu cs_1 cs_1')(\nu sr_1 sr_1')(\nu ss')(\nu rr')(Cliente \mid Servidor \mid RPI)$ e o proceso rotulado $Rpi_1 : \mathtt{lin}\ y(file).\overline{r'} sr_1'.\overline{sr_1} file.\mathtt{lin}\ sr_1(resp).\overline{y} resp.\mathbf{0}$, a condição de linearidade para (sr, sr') não seria mantida ao longo da execução. Por ocorrerem como a continuação do canal irrestrito s, teríamos a ocorrência dos mesmos a cada cópia a partir do $Servidor$, provocada pela conexão de um cliente. Uma nova proposta é então apresentada (Fig. 7), adequando a representação para uma utilização do canal compartilhado r de acordo ao sistema de tipos.

Sobre os tipos de sessão dessa representação, no processo do servidor temos agora a informação do tipo recursivo μs, determinando que o processo é replicado a cada uso. Nesse caso, o tipo de sessão do servidor nos fornece a informação de que o canal compartilhado s espera receber um canal com um tipo complementar ao canal privado do cliente,

$$P = (\nu cs_1 cs_1')(\nu ss')(\nu rr')(Cliente \mid Servidor \mid RPI)$$
$$Servidor = \texttt{un}\ s(y).($$
$$y \triangleright \{Rpi_1 : (\nu sr_1 sr_1')\texttt{lin}\ y(file).\overline{r'}sr_1'.\overline{sr_1}file.\texttt{lin}\ sr_1(resp).\overline{y}resp.\mathbf{0}\})$$
$$Cliente = \overline{s'}cs_1'.cs_1 \triangleleft Rpi_1.\overline{cs_1}\texttt{true}.\texttt{lin}\ cs_1(x).\mathbf{0}$$
$$RPI = \texttt{un}\ r(z).\texttt{lin}\ z(file)\overline{z}\texttt{true}.\mathbf{0}$$

Figura 7. Proposta 2

fazendo em seguida uma replicação do processo e voltando ao seu estado inicial. Esse canal enviado através de s' para s será o meio por onde o cliente pode manter a conexão com o servidor sem a interferência de outros clientes/processos. Com essa representação foi possível derivar as informações sobre o tipo dos processos para aplicar a funções de checagem de tipos [Restrepo et al. 2023]:

$$Servidor : \mu s\ \texttt{un}\ ?(\texttt{lin}\ \&\{(Rpi_1 : \texttt{lin}\ ?bool.(\texttt{lin}\ !bool.end))\}).\ s$$
$$Cliente : \texttt{lin}\ \oplus \{(Rpi_1 : \texttt{lin}\ !bool.(\texttt{lin}\ ?bool.end))\}$$
$$RPI : \mu r\ \texttt{un}\ ?(\texttt{lin}\ ?bool.(\texttt{lin}\ !bool.end)).\ r$$

Contudo, ao aplicar a função de checagem foi retornado o resultado `ill-typed-process`. Após uma análise do comportamento da checagem de tipos especificada em Maude, foi identificado que em um certo ponto da derivação, canais lineares que deveriam ser removidos eram inseridos em duplicidade no conjunto de canais L, utilizados na versão algorítmica do sistema de tipos (veja Fig. 10). Investigando as definições do conjunto de canais foi identificado que a equação responsável por remover duplicidades estava desabilitada (comentada) e, analisando a definição do construtor do conjunto de canais, o mesmo não possuía a propriedade de comutatividade (apenas associatividade e identidade foram definidas). Dessa forma, a representação do conjunto de canais atuava como uma lista e, ao aplicar a checagem, era inferido um resultado incorreto.

Após as correções na especificação em Maude, a representação na Fig. 7 com os respectivos tipos de sessão discutidos acima foram checados como bem-tipados pela ferramenta. Apesar de uma noção de linearidade presente no sistema de tipos, termos bem-tipados não têm a garantia de estarem livres de *deadlocks* ou *locks*. Porém, em [Restrepo et al. 2023] é introduzida uma forma de fazer essa checagem descrevendo a propriedade como uma fórmula da lógica temporal linear (LTL) e utilizando um *model checker*. Curiosamente, na primeira tentativa de aplicação dessa abordagem ocorreu um *loop* de execução, até que toda memória disponível para o programa seja utilizada fazendo com que o sistema operacional encerrasse a execução. Ao investigar o problema descobrimos que essa falha de execução se dá por causa das modificações feitas anteriormente na definição dos conjuntos de canais. Vale ressaltar que não é necessário que todas essa modificações sejam desfeitas para o funcionamento correto do *checker*, bastando desfazer uma das modificações (remover a propriedade de comutatividade ou comentar a equação novamente).

Ainda foram exploradas algumas variações das representações apresentadas, por exemplo, caso de um cliente e dois *Raspberry Pi's*, caso de dois clientes e apenas um *Raspberry Pi* e também dois clientes e dois *Raspberry Pi's*. O caso onde dois clientes se

$$P = (\nu cs_1 cs_1')(\nu cs_2 cs_2')(\nu ss')(\nu r_1 r_1')(\nu r_2 r_2')(\\ Cliente_1 \mid Cliente_2 \mid Servidor \mid RPI_1 \mid RPI_2)$$

$$Servidor = \texttt{un}\ s(y).(y \triangleright \{\\ (Rpi_1 : (\nu sr_1 sr_1')\texttt{lin}\ y(file).\overline{r_1'} sr_1'.\overline{sr_1} file.\texttt{lin}\ sr_1(resp).\overline{y} resp.\mathbf{0}),\\ (Rpi_2 : (\nu sr_2 sr_2')\texttt{lin}\ y(file).\overline{r_2'} sr_2'.\overline{sr_2} file.\texttt{lin}\ sr_2(resp).\overline{y} resp.\mathbf{0})\})$$

$$Cliente_i = \overline{s'} cs_i'.cs_i \triangleleft Rpi_i.\overline{cs_i}\texttt{true}.\texttt{lin}\ cs_i(x).\mathbf{0}$$

$$RPI_i = \texttt{un}\ r_i(z).\texttt{lin}\ z(file).\overline{z}\texttt{true}.\mathbf{0}$$

Figura 8. Proposta 3

conectam com dois *Raspberry Pi's* distintos é apresentado na Fig. 8. Para os testes com um cliente e dois *Raspberry Pi's* ou dois clientes e dois *Raspberry Pi's* foi possível aplicar a checagem de tipos. Porém, ao testar o caso com dois clientes e apenas um *Raspberry Pi* a ferramenta entre em um *loop* de execução, onde o problema ainda não foi identificado.

5. Conclusão e Trabalhos Futuros

O presente trabalho apresentou um estudo de caso da representação de um programa em Go como um termo em $\mathtt{s}\pi$. Alguns problemas na especificação em Maude apresentada em [Restrepo et al. 2023] foram identificados, enquanto outros dependem ainda de uma análise mais aprofundada. Por outro lado, dois aspectos relevantes em programas Go utilizando canais, não tratado no presente trabalho, que são a comunicação assíncrona e a possibilidade de *time-outs* não podem ser representados em $\mathtt{s}\pi$. Nesse caso, o sistema de [Bocchi et al. 2019] seria uma possibilidade em investigações futuras.

Trabalhos Relacionados Em [Lange et al. 2017] uma abordagem análoga é apresentada na utilização de *behavioural types* na garantia de processos livres de erros de comunicação e *locks*, onde programas em Go são transpilados para um cálculo de processos com a inferência de tipos decidível, onde os tipos satisfazendo uma restrição – chamados de *fenced types*– permitem a checagem dessas propriedades. A possibilidade de *time-outs* é tratada com transições silenciosas –utilizando o prefixo $\tau.P$– divergindo da abordagem em [Bocchi et al. 2019]. Assim, uma comparação das classes de programas bem-tipados em cada caso seria uma investigação interessante. Mais recentemente, foram apresentadas abordagens na utilização dos tipos de sessão em outra direção. Em [Echarren Serrano 2020] é apresentada uma ferramenta para verificar a especificação de protocolos, baseado em tipos de sessão multipartidários [Demangeon and Honda 2012], fazendo a extração de APIs em Go para os protocolos verificados. Por outro lado, em [Geraldo 2022] é apresentada uma linguagem funcional com suporte a concorrência, com tipos de sessão relacionados a lógica linear intuicionista (ILL), onde um programa bem-tipado é compilado para um programa em Go. Em ambos os casos, o usuário deve utilizar uma linguagem/ferramenta diversa para a obtenção dos artefatos em Go, enquanto a presente proposta segue a abordagem em [Lange et al. 2017], em auxiliar o programador em Go na obtenção de programas (concorrente) corretos.

Referências

Bocchi, L., Murgia, M., Vasconcelos, V. T., and Yoshida, N. (2019). Asynchronous timed session types - from duality to time-sensitive processes. In *ESOP*, volume 11423 of *Lecture Notes in Computer Science*, pages 583–610. Springer.

Demangeon, R. and Honda, K. (2012). Nested protocols in session types. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer.

Donovan, A. A. and Kernighan, B. W. (2015). *The Go programming language*. Addison-Wesley Professional.

Echarren Serrano, B. (2020). *Nested Multiparty Session Programming in Go.* Master's thesis, Imperial College London. Disponível em: `https://becharrens.files.wordpress.com/2020/07/final_report.pdf`.

Geraldo, J. M. P. D. C. R. (2022). *Making Session Types Go*. Master's thesis, NOVA University Lisbon. Disponível em: `http://ctp.di.fct.unl.pt/~btoninho/student-theses/geraldo22-msthesis.pdf`.

Go Language (2009). Effective Go. Disponível em: `https://go.dev/doc/effective_go`. Último acesso em 17 de Agosto de 2023.

Kobayashi, N. (2002). Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer.

Lange, J., Ng, N., Toninho, B., and Yoshida, N. (2017). Fencing off Go: liveness and safety for channel-based programming. In *POPL*, pages 748–761. ACM.

Restrepo, C. A. R., Jaramillo, J. C., and Pérez, J. A. (2023). Session-based concurrency in maude: Executable semantics and type checking. *Journal of Logical and Algebraic Methods in Programming*, 133:100872.

Vasconcelos, V. T. (2012). Fundamentals of session types. *Information and Computation*, 217:52–70.

A. Tipos de Sessão Algorítmico

$$\Gamma \div \emptyset = \Gamma \qquad \frac{\Gamma_1 \div L = \Gamma_2, x : T \qquad \mathtt{un}(T)}{\Gamma_1 \div (L, x) = \Gamma_2} \qquad \frac{\Gamma_1 \div L = \Gamma_2 \qquad x \notin \mathtt{dom}(\Gamma_2)}{\Gamma_1 \div (L, x) = \Gamma_2}$$

Figura 9. Regras para diferença de contextos

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{bool}; \Gamma}\ [\text{A-TRUE}] \qquad \frac{}{\Gamma_1, x : \mathtt{lin}\ p, \Gamma_2 \vdash x : \mathtt{lin}\ p; (\Gamma_1, \Gamma_2)}\ [\text{A-LINVAR}]$$

$$\frac{}{\Gamma \vdash \mathtt{false} : \mathtt{bool}; \Gamma}\ [\text{A-FALSE}] \qquad \frac{\mathtt{un}(\Gamma)}{\Gamma_1, x : T, \Gamma_2 \vdash x : T; (\Gamma_1, x : T, \Gamma_2)}\ [\text{A-UNVAR}]$$

$$\frac{}{\Gamma \vdash \mathbf{0} : \Gamma; \emptyset}\ [\text{A-INACT}] \qquad \frac{\Gamma_1 \vdash P : \Gamma_2; L_1 \qquad \Gamma_2 \div L_1 \vdash Q : \Gamma_3; L_2}{\Gamma_1 \vdash P \mid Q : \Gamma_3; L_2}\ [\text{A-PAR}]$$

$$\frac{\Gamma_1, x : T, y : \overline{T} \vdash P : \Gamma_2; L}{\Gamma_1 \vdash (\nu xy : T)P : \Gamma_2 \div \{x, y\}; L \setminus \{x, y\}}\ [\text{A-RES}]$$

$$\frac{\Gamma_1 \vdash x : q!T.U; \Gamma_2 \qquad \Gamma_2 \vdash v : T; \Gamma_3 \qquad \Gamma_3 + x : U \vdash P : \Gamma_4; L}{\Gamma_1 \vdash \overline{x}v.P : \Gamma_4; L \cup (\mathtt{if}\ q = \mathtt{lin}\ \mathtt{then}\ \{x\}\ \mathtt{else}\ \emptyset)}\ [\text{A-OUT}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash x : q_2?T.U; \Gamma_2 \\ (\Gamma_2, y : T) + x : U \vdash P : \Gamma_3; L \qquad q_1 = \mathtt{un} \Rightarrow L \setminus \{y\} = \emptyset\end{array}}{\Gamma_1 \vdash q_1 x(y).P : \Gamma_3 \div \{y\}; L \setminus \{y\} \cup (\mathtt{if}\ q_2 = \mathtt{lin}\ \mathtt{then}\ \{x\}\ \mathtt{else}\ \emptyset)}\ [\text{A-IN}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash x : q\&\{l_i : T_i\}_{i \in I}; \Gamma_2 \quad \Gamma_2 + x : T_i \vdash P_i : \Gamma_3; L_i \\ \forall_{i \in I, j \in I}\ L_i \setminus \{x\} = L_j \setminus \{x\}\end{array}}{\Gamma_1 \vdash x \triangleright \{l_i : P_i\}_{i \in I} : \Gamma_3; L \cup (\mathtt{if}\ q = \mathtt{lin}\ \mathtt{then}\ \{x\}\ \mathtt{else}\ \emptyset)}\ [\text{A-BRANCH}]$$

$$\frac{\Gamma_1 \vdash x : q \oplus \{l_i : T_i\}_{i \in I}; \Gamma_2 \quad \Gamma_2 + x : T_j \vdash P : \Gamma_3; L \quad j \in I}{\Gamma_1 \vdash x \triangleleft l_j.P : \Gamma_3; L \cup (\mathtt{if}\ q = \mathtt{lin}\ \mathtt{then}\ \{x\}\ \mathtt{else}\ \emptyset)}\ [\text{A-SEL}]$$

Figura 10. Regras de Checagem Algorítmica de Tipos