
Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

# Detección estática de canales no cerrados usando tipos comportamentales

Tesis de Licenciatura en Ciencias de la Computación

Daniela Marottoli

Director: Hernán Melgratti

Buenos Aires, 2019

**Detección estática de canales no cerrados usando tipos comportamentales**

Go es un lenguaje de programación tipado estáticamente cuyo diseño presenta primitivas explícitas de intercambio de mensajes a través de canales. Lange, Ng, Toninho y Yoshida proponen un sistema de tipos comportamentales para analizar propiedades sobre aspectos comunicacionales de programas en Go, tales como ausencia de *deadlocks* y errores de comunicación. Esta verificación se realiza a través de dos herramientas, Dingo Hunter y Gong, que toman un programa en Go e indican si satisfacen estas propiedades. Furman extiende estas herramientas agregando mensajes de error que contienen información sobre la línea de código que viola las propiedades, y la historia de ejecución. El objetivo de este trabajo es mejorar la usabilidad de las herramientas incorporando una detección estática de canales abiertos, de manera de agregar en el *output* información sobre canales que no son cerrados y que no se utilizan en el futuro de la ejecución. Para ello se extiende la implementación de Gong de manera de obtener información sobre tipos asociados a canales que se eliminan cuando se construye la máquina de estados que representa la ejecución simbólica de los tipos comportamentales de un programa.

**Palabras claves:** Tipos comportamentales, Go, Comunicación por intercambio de mensajes, Cierre de canales, Deadlock, Liveness.

# AGRADECIMIENTOS

A mis viejxs que se rompieron el lomo por mi bienestar, por mi educación, por mi futuro. Sin su apoyo jamás podría haber transitado una carrera como ésta.

A mis amigas que fueron mi contención emocional durante estos 10 años.

A mis compañerxs de militancia que me enseñaron a ser una mejor persona y a luchar por lo que creo justo. Y con quienes comparto el sueño de lograr una universidad más inclusiva, un sistema científico soberano, y un país para todxs.

A mis compañerxs de la facultad con lxs que transitamos este hermoso pero sacrificado camino que implica estudiar en Exactas.

A todxs lxs miembrxs del laboratorio LaFHIS que me hicieron conocer el universo científico. Trabajar con ellxs además me permitió descubrir mi amor por la docencia e involucrarme en el DC.

A Néstor y a Cristina, que hicieron que me enamore de la política. Ellxs no solo me dieron un contexto político, social y económico que me permitió estudiar, sino que me hicieron replantear mi rol social como estudiante universitaria. “Ser universitarix, más que un privilegio, es un deber de compromiso con el pueblo”.

Y por último, y siendo el 70 aniversario de la gratuidad universitaria decretado por Perón en 1949, agradezco tener en mi país a la mejor universidad de Latinoamérica, y que ésta sea pública, gratuita y de calidad. “La conquista más grande fue que la Universidad se llenó de hijxs de obrerxs, donde antes estaba solamente admitido el oligarca”.

# Índice general

# 1. INTRODUCCIÓN

La importancia de la programación basada en la comunicación de procesos se incrementó sustancialmente durante los últimos años. Desde protocolos de red a través de Internet para sistemas cliente-servidor hasta aplicaciones distribuidas, asegurar una correcta interacción entre los actores que se comunican se volvió un elemento fundamental para lograr el buen funcionamiento de las aplicaciones. Por este motivo surgen sistemas de tipado llamados *tipos comportamentales* o *tipos sesión*[1], que describen el comportamiento de los actores que participan en una comunicación, y asegura que un programa bien tipado cumple determinadas propiedades, por ejemplo, la ausencia de *deadlocks*. En este trabajo se utilizarán tipos comportamentales para analizar el comportamiento de programas escritos en Go.

Go es un lenguaje de programación tipado estáticamente que surge en el año 2007, cuyo diseño presenta primitivas explícitas de intercambio de mensajes a través de **canales** (en vez del uso de memoria compartida), lo que lo convierte en un lenguaje especialmente apropiado para la programación concurrente[2]. Además incorpora el concepto de **go-rutina**, que es un proceso liviano que se ejecuta en paralelo y se comunica con otras go-rutinas intercambiando mensajes a través de canales. Go hereda la mayoría de los problemas que se encuentran comunmente en la programación concurrente, ofreciendo pocas garantías de una correcta estructura de la comunicación en tiempo de compilación. Por este motivo es un gran aporte proveer un sistema de tipos comportamentales para Go, ya que puede prevenir errores de comunicación en los programas antes de ejecutarlos.

El trabajo realizado por Lange, Ng, Toninho y Yoshida[3] propone un *framework* para la verificación estática de ausencia de *deadlocks* y de errores de comunicación (propiedades de *liveness* y *safety*) de programas en Go extrayendo tipos comportamentales concurrentes del código fuente. Los tipos pueden verse como una representación abstracta del comportamiento comunicacional del programa. Luego se realiza un análisis de la ejecución simbólica de los tipos comportamentales para verificar las propiedades. Este *framework* está compuesto por dos herramientas: **Dingo Hunter**[4], que traduce el código original a un lenguaje intermedio llamado MiGo, y **Gong**[5], que genera los tipos comportamentales a partir del código en MiGo y devuelve dos valores *booleanos* indicando si cumplen las propiedades de *safety* y *liveness*.

Furman extiende ambas herramientas[6] con el objetivo de agregar más información al resultado. En lugar de indicar sólo si las propiedades se satisfacen o no (es decir, dos valores de verdad), se le agregan mensajes de error indicando en qué línea del código ori-

ginal se encuentra la instrucción que viola las propiedades y cómo se llegó a ese estado de la ejecución. Para lograr esto se le agrega a los tipos comportamentales la información adicional de la historia de ejecución previa.

El objetivo del siguiente trabajo consiste en asegurar que un programa en Go cumpla uno de los principios de comunicación fundamentales para el buen funcionamiento: el **principio de cierre de canales**[7]. Éste establece que, en una comunicación entre go-rutinas sobre un canal, la go-rutina emisora debe cerrar el canal para indicar que no se enviarán más valores sobre el mismo. Este trabajo propone extender la herramienta **Gong** con un verificador estático de canales abiertos, utilizando la estructura de tipos comportamentales y analizando su ejecución simbólica, de manera de identificar las líneas del código en donde se crean canales que no son cerrados y que no se vuelven a utilizar.

## 1.1. Contribuciones

Detallaremos a continuación las contribuciones de este trabajo:

- Definimos formalmente la noción de tipos cuya ejecución no cierra canales inutilizados.
- Proponemos un algoritmo para decidir si un tipo no cierra canales inutilizados durante la ejecución. El mismo se define analizando el LTS que representa la ejecución simbólica del tipo comportamental.
- Adaptamos la implementación de Gong de manera tal que sea posible analizar si un programa Go no cierra canales inutilizados.
- Discutimos las limitaciones de Gong para analizar programas que usan ciclos con `range`.

## 1.2. Organización del material

En el siguiente capítulo se resumen los antecedentes de conceptos que utilizaremos a lo largo del trabajo. Estos son los tipos comportamentales, cómo funciona la comunicación en Go, la representación SSA y las propiedades *liveness* y *safety*. En el capítulo 3 brindamos detalles sobre la construcción del lenguaje MiGo, de los tipos comportamentales elegidos para representarlo, y de las herramientas Dingo Hunter y Gong que pueden ser de utilidad para la presentación de las secciones restantes. En el capítulo 4 se detalla el algoritmo de detección de canales abiertos inutilizables y su implementación en Gong. Luego se explican algunas limitaciones que tienen las herramientas desarrolladas en [3] con respecto al análisis de programas que cumplen ciertas características. Finalmente en el capítulo 6 se presentan consideraciones finales y se discuten posibles extensiones del trabajo realizado.

# 2. ANTECEDENTES

## 2.1. Tipos comportamentales

Los sistemas de software basados en la cooperación y comunicación de aplicaciones distribuidas se han convertido en una herramienta fundamental de la sociedad moderna, y las consecuencias de las fallas de este tipo de aplicaciones pueden ser muy graves. Estas fallas se deben frecuentemente a problemas en la forma de integrar las distintas componentes del sistema, es decir, a establecer correctos protocolos de interacción. El desafío tecnológico para que tal integración no tenga fallas, entonces, presupone razonar sobre los comportamientos de tales sistemas y las propiedades que son garantizadas por la composición.

Una propiedad de particular interés para las aplicaciones centradas en la comunicación es la correcta terminación de todos los componentes que participan en la interacción, y a partir de esta necesidad surge la teoría de los **tipos comportamentales**. La idea detrás de este enfoque es asociar un tipo a cada servicio. El tipo es una descripción abstracta del comportamiento visible de cada servicio. Luego una verificación estática a nivel de tipos determina si los servicios son compatibles o no, es decir, si el análisis de los tipos determina que los mismos son compatibles, esto asegura que el sistema compuesto funciona correctamente. Los contratos o tipos son básicamente procesos en algún lenguaje de procesos concurrentes, tales como CCS[8], que describen el orden en que cada servicio invoca o acepta invocaciones de otros servicios. Estas operaciones se modelan abstractamente como acciones de entrada y salida que tienen lugar a través de canales o nombres.

En este trabajo se definirán tipos comportamentales específicos para la comunicación de los programas en Go. El objetivo es codificar la estructura correcta de comunicación entre distintos *threads* para verificar distintas propiedades sin necesidad de ejecutar el programa, ya que los tipos cumplen el rol de procesos y se pueden ejecutar simbólicamente para lograr tal fin.

## 2.2. Comunicación en Go

Los modelos de comunicación tradicionales (usados en lenguajes como Java o C++) utilizan *threads* que se comunican a través de memoria compartida. En general, las estructuras de datos compartidas son protegidas por *locks* (bloqueos), y los *threads* solo pueden acceder a los datos si no están bloqueados.

La comunicación en Go está basada fuertemente en el cálculo de procesos CSP[13] (Comunicación Secuencial de Procesos), y se modela a través de **canales** y **go-rutinas**, proporcionando así una manera distinta y elegante de estructurar la programación concurrente. En lugar de usar explícitamente *locks* para mediar el acceso a datos compartidos, en Go pueden usarse canales para pasar referencias a datos entre go-rutinas que corren en paralelo, garantizando que solo una go-rutina tenga acceso a los datos en un momento dado, y que, por lo tanto, no exista el problema de las *race-conditions*[9]. Es por este motivo que Go tiene un lema que dice *"Do not communicate by sharing memory; instead, share memory by communicating"*.

Una go-rutina es una función que se ejecuta en paralelo con otras go-rutinas y que son administradas por el *scheduler* de Go. Es liviana y cuesta poco más que la asignación de espacio de pila. Y las pilas comienzan siendo pequeñas, por lo que son baratas, y crecen asignando y liberando espacio dinámicamente según sea necesario. Las go-rutinas se multiplexan en muchos subprocesos del sistema operativo, por lo que si uno debe bloquearse, otros continúan ejecutándose. Su diseño oculta muchas de las complejidades de la creación y mantenimiento de *threads*[10].

Las go-rutinas se invocan a través del comando `go` seguido de una llamada a una función. Esto genera que la función se ejecute en un hilo paralelo a la ejecución principal, aunque sigue teniendo el mismo contexto.

Los canales funcionan como "conductos" a través de los cuales se comunican las go-rutinas, y se pasan valores de cierto tipo. Al igual que las listas y los diccionarios, se crean con la primitiva `make`, y el valor resultante actúa como una referencia a la estructura de datos subyacente. Se debe proporcionar un parámetro que indique el tipo del canal, y un parámetro de tipo entero opcional, que establece el tamaño del *buffer* (el valor predeterminado si no se ingresa el segundo parámetro es cero, y define un canal sin *buffer* o sincrónico). Así, un canal sincrónico de tipo entero se crea mediante la instrucción `make(chan int)`[2]. No hay un límite en la cantidad de go-rutinas que pueden utilizar un canal en simultáneo. Distintas go-rutinas pueden compartir un canal a través del acceso común a la variable que lo referencia, o mediante el pasaje de la referencia como parámetro de función.

Un canal tiene dos operaciones principales: *send* y *receive*. La instrucción `send` transmite un valor de una go-rutina, a través del canal, a otra go-rutina que ejecutó el `receive` correspondiente. Ambas operaciones se escriben usando el operador `<-`. En una declaración de envío, el `<-` separa los operandos de canal y valor, mientras que en la recepción, el `<-` precede al operando del canal:

```
ch <- x     // enviar x por el canal ch
```

```
x = <- ch   // recibir el valor de ch y guardarlo en x
<- ch       // recibir el valor de ch; el resultado se descarta
```

El funcionamiento de la comunicación basada en canales sincrónicos es el siguiente: cuando una go-rutina intenta enviar o recibir de un canal, se bloquea hasta que otra go-rutina intenta la operación de recepción o envío correspondiente, en cuyo punto el valor es transferido y ambas proceden con el resto de la ejecución. Esto es lo que llamamos **sincronización**.

Un canal con *buffer* tiene una cola de elementos (implementada como una cola FIFO), cuya capacidad máxima se determina cuando es creado. El `send` inserta un elemento al final de la cola, y el `receive` elimina un elemento del principio. Si el canal está lleno, el `send` bloquea su go-rutina hasta que se haga espacio en la cola por el `receive` de otra go-rutina. Análogamente, si el canal está vacío, el `receive` bloquea hasta que haya un `send`.

Los canales también soportan una tercera operación llamada *close*, que activa un *flag* indicando que el canal está cerrado y no se podrán enviar más valores. Si se intenta enviar un mensaje a través de un canal cerrado, se producirá una *panic failure* y se detendrá por completo la ejecución del programa. Lo mismo sucede si se intenta cerrar un canal que ya ejecutó su operación `close`. Sin embargo puede realizarse un `receive` de un canal cerrado, en donde se devuelven los valores que quedaron en el *buffer* en el caso de un canal asincrónico, o el valor por defecto del tipo del canal (0 para los enteros, *false* para los booleanos, etc).

Otra de las piezas claves que distinguen a Go como un lenguaje ideal para la programación concurrente es el comando `select`. Éste codifica una elección no determinística en donde cada guarda es una acción de comunicación, y se ejecuta la primera rama que se pueda siempre y cuando haya otra acción con la que pueda sincronizar. Cuando se pueden elegir varias ramas simultáneamente, se elige una al azar (a través de la generación de números pseudoaleatorios). Existe una acción que siempre se puede ejecutar en las guardas del `select`. Se llama `tau` y representa una acción silenciosa que no tiene efectos en cuanto al comportamiento comunicacional.

En la figura 2.1 podemos observar un ejemplo sencillo de un programa en Go que calcula el factorial de 5. Tiene un método principal (`main`) que crea un canal sincrónico `a`, y lanza una go-rutina de la función `fact` pasándole como argumento el número que representa el factorial que se quiere calcular y el canal que acaba de crear. Luego imprime por pantalla lo que recibe de `a`. El método `fact` verifica si está en el caso base (si el número es 1) y en ese caso lo envía por el canal que recibe como argumento y termina la ejecución. Si no está en el caso base, crea un canal `b2`, y lanza una go-rutina recursiva que va a

```
1   func main() {
2       a := make(chan int)
3       go fact(5, a)
4       fmt.Println(<-a)
5   }

6   func fact(n int, b chan<- int) {
7       if n <= 1 {
8           b <- n
9           return
10      }
11      b2 := make(chan int)
12      go fact(n-1, b2)
13      b <- n * <-b2
14  }
```

*Fig. 2.1:* Ejemplo de un programa en Go

enviar sobre `b2` el factorial del número anterior. Finalmente envía sobre `b` el resultado de multiplicar a `n` por el factorial de `n-1`.

## 2.3. Static single assignment (SSA)

SSA es una librería de Go que define una representación intermedia estática y de asignación simple de los elementos de un programa en Go (tipos, funciones, variables, constantes, etc). La utilidad principal de SSA proviene de cómo simplifica y mejora los resultados de varias optimizaciones del compilador, como la propagación de constantes, eliminación de código muerto, asignación de registros, etc[11].

SSA introduce muchos cambios para simplificar el código fuente, pero principalmente, como su nombre lo indica, se centra en lograr una simplificación en las propiedades de las variables, de manera que éstas sean asignadas únicamente una vez. A efectos de este trabajo se destaca cómo parsea las estructuras de control de flujo: los ciclos (`for` y `while`) son reemplazados por comandos `jump` y condicionales. Consideremos el ejemplo de la figura 2.2. Este es un programa sencillo que cicla infinitiamente enviando y recibiendo valores del canal `ch`.

En la figura 2.3 vemos el resultado de extraer la representación SSA de la figura 2.2. Sin entrar en detalles sobre la sintaxis del código, es interesante remarcar cómo son parseados los ciclos del programa original. A los bloques de instrucciones se les asigna una etiqueta, y aquellas que forman parte del cuerpo del `for` terminan con la instrucción `jump` con su

```
func main () {
    ch := make ( chan int )
    go recv (ch)
    for {
        ch <- 1
    }
}

func recv (ch chan int) {
    for {
        <-ch
    }
}
```

*Fig. 2.2:* Ejemplo de ciclos de un programa en Go

etiqueta correspondiente. Es importante destacar que los ciclos del ejemplo son infinitos, por lo que no existen condiciones para salir de los mismos. Veremos más adelante que SSA no es muy útil para representar ciclos acotados y esto representa un problema a la hora validar propiedades estáticamente sobre un programa en Go.

## 2.4. Liveness y Safety

Si bien el sistema de tipos de Go garantiza que los canales se utilicen para comunicar valores del tipo apropiado, no puede ofrecer garantías sobre propiedades de *liveness* y *safety* de un programa de manera estática.

Por ejemplo, una propiedad de *liveness* sería asegurar que las acciones de comunicación de un programa se ejecuten tarde o temprano. Es decir, debe haber un progreso tanto de la ejecución principal como de todos los hilos de ejecución, lo que garantiza la ausencia de *deadlocks* totales y parciales.

Una propiedad de *safety* de un programa podría asegurar la ausencia de errores de comunicación. En particular en Go, podemos identificar dos errores sobre canales en tiempo de ejecución que se corresponden con cerrar un canal más de una vez, o enviar un mensaje sobre un canal cerrado.

```
func main():
0:                                                          entry P:0 S:1
    t0 = make chan int 0:int                                     chan int
    ; *ast.CallExpr @ 4:8 is t0
    ; var ch chan int @ 4:2 is t0
    ; func main.recv(ch chan int) @ 5:5 is recv
    ; var ch chan int @ 5:11 is t0
    go recv(t0)
    jump 1
1:                                                        for.body P:2 S:1
    ; var ch chan int @ 7:3 is t0
    send t0 <- 1:int
    jump 1

func recv(ch chan int):
0:                                                          entry P:0 S:1
    jump 1
1:                                                        for.body P:2 S:1
    ; var ch chan int @ 13:5 is ch
    t0 = <-ch                                                        int
    ; *ast.UnaryExpr @ 13:3 is t0
    jump 1
```

*Fig. 2.3:* Representación SSA para el código de la figura 2.2

# 3. FENCING OFF GO

El trabajo realizado en [3] tiene como objetivo realizar un chequeo estático de *liveness* y *safety* para programas en Go, a través de la validación de propiedades en la ejecución simbólica de tipos comportamentales. Para ello se propone establecer un lenguaje intermedio llamado **MiGo** (mini-go) cuyo objetivo es abstraer la información básica de comunicación del programa original, y a partir de éste generar los tipos correspondientes.

Los tipos comportamentales tienen una semántica operacional y por lo tanto pueden ejecutarse simbólicamente. Sin embargo, muchos programas pueden ser cíclicos o recursivos sin terminación, por lo cual es necesario considerar una cantidad finita de ejecuciones. Es por esto que el trabajo considera solo los tipos *fenced*. Si un tipo es *fenced*, entonces durante su ejecución solo puede haber un conjunto finito de canales compartidos por una cantidad finita de tipos (hilos de ejecución). Esto garantiza que el programa tenga una cantidad finita de patrones de comunicación (que podrían repetirse infinitamente), y por lo tanto, su ejecución simbólica termine.

## 3.1. MiGo

El leguaje MiGo está basado en el cálculo de procesos secuenciales (CSP)[13], y modela el comportamiento comunicacional basado en canales de un programa en Go. Su sintaxis puede verse en la figura 3.1.

```
P,Q := π; P
     | close u; P
     | select{π_i ; P_i}_{i∈I}
     | if e then P else Q
     | newchan(y : σ); P
     | P | Q
     | 0
     | X⟨ẽ, ũ⟩
     | (νc)P
     | c⟨σ⟩ :: ṽ | c*⟨σ⟩ :: ṽ
```

```
u   := a | x
π   := u!⟨e⟩ | u?(y) | τ
v   := n | true | false | x
e   := v | not(e) | succ(e)
D   := X(x̃) = P
P   := {D_i}_{i∈I} in P
σ   := bool | int | ...
```

*Fig. 3.1:* Sintaxis de MiGo

$P$ y $Q$ representan procesos, $\pi$ son primitivas de comunicación, $e$ son expresiones, $x$ e $y$ variables, y $\tilde{u}$ y $\tilde{x}$ representan lista de expresiones y variables, respectivamente.

El lenguaje se construye de la siguiente manera: el proceso $\pi; P$ denota la acción $\pi$

seguida del proceso $P$. Una acción $\pi$ puede ser $u!\langle e\rangle$ que envía $e$ por el canal $u$, $u?(y)$ que recibe del canal $u$ y almacena en la variable $y$, o una acción silenciosa $\tau$. `close` $u; P$ cierra el canal $u$ y continúa $P$. `select`$\{\pi_i\,; P_i\}_{i\in I}$ denota una elección no determinística entre los procesos $P_i$, donde cada $P_i$ tiene la guarda $\pi_i$. `if` $e$ `then` $P$ `else` $Q$ es el condicional estándar, $P \mid Q$ la composición en paralelo entre procesos, y $\mathbf{0}$ el proceso inactivo (suele omitirse). `newchan`$(y : \sigma)$ crea un nuevo canal de tipo $\sigma$ y lo liga con $y$ en el siguiente proceso $P$. $X\langle \tilde{e}, \tilde{u}\rangle$ denota la llamada a una función $X$ con los parámetros $\tilde{e}$ y $\tilde{u}$. Por último, $(\nu c)P$ y *buffers* son constructores en tiempo de ejecución (es decir, no son escritos explícitamente por el programador): el primero declara el nombre de un canal en un proceso, y $c\langle\sigma\rangle :: \tilde{v}$ es un *buffer* para un canal abierto que contiene mensajes $\tilde{v}$ de tipo $\sigma$ ($c^\star\langle\sigma\rangle$ denota un canal cerrado).

Un programa en Go va a estar definido como un programa **P** de MiGo. **P** se escribe como $\{D_i\}_{i\in I}$ `in` $P$, consiste en un conjunto de definiciones de procesos $D$ que codifican todas las funciones y las go-rutinas utilizadas en el programa, junto con el proceso $P$ que codifica el punto de entrada (lo que en Go es la función `main`). Puede observarse en la figura 3.2 el código en MiGo para el programa en Go definido en la figura 2.1.

$F(n, c) \triangleq$ `if` $(n \leq 1)$ `then` $c!\langle n\rangle$
`else newchan`$(ch1 :$ `int`$)$; $(F(n-1, ch1) \mid ch1?(x);\, c!\langle n * x\rangle)$

P $\triangleq \{F(n, c)\}$ `in newchan`$(c :$ `int`$)$; $(F(5, c) \mid c?(x); 0)$

*Fig. 3.2:* Representación en MiGo para el programa de la figura 2.1

Luego se definen las reglas de reducción para la semántica operacional ($P \to Q$) de MiGo en la figura 3.3 (muy similar a la de Go). Para simplificar la definición, se trabajará con canales sincrónicos (es decir, con un *buffer* de tamaño 0). La regla [SCOM] especifica la sincronización entre una acción de envío y otra de recepción. [SCLOSE] define la recepción del valor por defecto $v^\sigma$ del tipo $\sigma$ sobre canales cerrados. La regla [CLOSE] cambia el estado de un *buffer* de abierto ($c\langle\sigma\rangle :: \tilde{v}$) a cerrado ($c^\star\langle\sigma\rangle :: \tilde{v}$). [NEWC] crea un canal fresco $c$, instanciándolo acordemente en $P$ y creando el *buffer* del canal. La regla [SEL] codifica la elección no determinística de algún subproceso $P_i$ que esté listo para reducir. [DEF] reemplaza $X$ por su correspondiente definición en $\{D_i\}_{i\in I}$ instanciando acordemente sus parámetros. El resto de las reglas son las estándar del cálculo de procesos.

$$[\text{SCOM}]\ \frac{e \downarrow v}{c!\langle e\rangle; P \mid c?(y); Q \mid c\langle\sigma\rangle :: \emptyset \ \rightarrow\ P \mid Q\{^{v}/_{y}\} \mid c\langle\sigma\rangle :: \emptyset}$$

$$[\text{SCLOSE}]\ \frac{}{c?(y); P \mid c^{\star}\langle\sigma\rangle :: \emptyset \ \rightarrow\ P\{^{v^{\sigma}}/_{y}\} \mid c^{\star}\langle\sigma\rangle :: \emptyset}$$

$$[\text{CLOSE}]\ \frac{}{\texttt{close}\ c; P \mid c\langle\sigma\rangle :: \tilde{v} \ \rightarrow\ P \mid c^{\star}\langle\sigma\rangle :: \tilde{v}}$$

$$[\text{TAU}]\ \frac{}{\tau; P \ \rightarrow\ P}$$

$$[\text{NEWC}]\ \frac{c \notin fn(P)}{\texttt{newchan}(y : \sigma);\ P \ \rightarrow\ (\nu c)(P\{^{c}/_{y}\} \mid c\langle\sigma\rangle :: \emptyset)}$$

$$[\text{PAR}]\ \frac{P \ \rightarrow\ P'}{P \mid Q \ \rightarrow\ P' \mid Q} \qquad [\text{RES}]\ \frac{P \ \rightarrow\ P'}{(\nu c)P \ \rightarrow\ (\nu c)P'}$$

$$[\text{STR}]\ \frac{P \equiv Q \ \rightarrow\ Q' \equiv P'}{P \ \rightarrow\ P'} \qquad [\text{SEL}]\ \frac{\pi_j; P_j \mid P \ \rightarrow\ R \quad j \in I}{\texttt{select}\{\pi_i; P_i\}_{i\in I} \mid P \ \rightarrow\ R}$$

$$[\text{IFT}]\ \frac{e \downarrow \texttt{true}}{\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \ \rightarrow\ P} \qquad [\text{IFF}]\ \frac{e \downarrow \texttt{false}}{\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \ \rightarrow\ Q}$$

$$[\text{DEF}]\ \frac{P\{^{\tilde{v},\tilde{c}}/_{\tilde{x}}\} \mid Q \ \rightarrow\ R \qquad e_i \downarrow v_i \qquad X(\tilde{x}) = P \in \{D_i\}_{i\in I}}{X(\tilde{e},\tilde{c}) \mid Q \ \rightarrow\ R}$$

*Fig. 3.3:* Semántica de MiGo

## 3.2. Sistema de Tipos Comportamentales para MiGo

Los tipos de los canales de Go sirven para determinar qué valores pueden ser enviados o recibidos a lo largo de un canal. El sistema de tipos definido en la figura 3.4 sirve como una abstracción del comportamiento de un programa válido de MiGo y permite validar propiedades estáticamente.

$$T, S \texttt{ := } \kappa; T \texttt{ | } \oplus\{T_i\}_{i\in I} \texttt{ | } \&\{T_i\}_{i\in I} \texttt{ | } (T \mid S) \texttt{ | } \texttt{0} \texttt{ | } \texttt{(new } a\texttt{)}T \texttt{ | } \texttt{end}[u]; T$$

$$\texttt{ | } t_X\langle\tilde{u}\rangle \texttt{ | } (\nu a)T \texttt{ | } \lfloor a \rfloor \texttt{ | } a^\star$$

$$\texttt{T := } \{t_{X_i}(\tilde{y}_i) = T_i\} \texttt{ in } S$$

$$\kappa \texttt{ := } \overline{u} \texttt{ | } u \texttt{ | } \tau$$

*Fig. 3.4:* Sintaxis de tipos

La sintaxis de los tipos $T, S$ es un reflejo de los procesos de MiGo: el tipo $\kappa; T$ denota enviar sobre un canal $u$, recibir de un canal $u$, o una acción silenciosa $\tau$, seguida por el comportamiento denotado por el tipo $T$. El tipo $\oplus\{T_i\}_{i\in I}$ representa la elección interna (`if-then-else`) entre los $T_i$, mientras que $\&\{T_i\}_{i\in I}$ denota la elección externa (`select`) entre los comportamientos $T_i$ dependiendo de las guardas $\kappa_i$. Los tipos incluyen composición en paralelo $T \mid S$, inacción `0` y creación de canales (`new` $a$)$T$ (ligando $a$ en $T$). El tipo `end`$[u]; T$ representa cerrar un canal $u$ seguido de un comportamiento $T$. El tipo $\{t_{X_i}(\tilde{y}_i) = T_i\}$ `in` $S$ codifica un conjunto de definiciones de tipos recursivos $\{t_{X_i}(\tilde{y}_i) = T_i\}$ ligados en $S$. Este conjunto de ecuaciones llamado `T` es el tipo asignado a los programas $\mathbf{P}$. La expresión $t_X\langle\tilde{u}\rangle$ denota la llamada a la definición recursiva de $t_X$ definida en el contexto, reemplazando los parámetros correspondientes en $\tilde{u}$. Por último, los tipos $(\nu a)T$, $\lfloor a \rfloor$ y $a^\star$ denotan ligar el canal $a$ en el comportamiento $T$, y los canales abiertos y cerrados respectivamente.

En la figura 3.5 podemos ver las reglas de tipado para procesos y programas de MiGo. En el juicio $\Gamma \vdash P \blacktriangleright T$ para procesos, $\Gamma$ representa el contexto, $P$ es un proceso, y $T$ es un tipo comportamental. Las reglas implementan una correspondencia bastante cercana entre los procesos y sus respectivos tipos: $\langle$IN$\rangle$ y $\langle$OUT$\rangle$ modelan que la recepción o el envío de mensajes a través de un canal $u$, seguidos de la ejecución $P$, tienen tipo $\overline{u}; T$ y $u; T$ respectivamente, siempre que $u$ sea un canal de tipo $\sigma$, el mensaje enviado o la variable sobre la que se recibe el mensaje sean también de tipo $\sigma$, y la continuación $P$ tenga tipo $T$. Las reglas $\langle$SEL$\rangle$ e $\langle$IF$\rangle$ tipan las elecciones no determinísticas (`select`) y determinísticas (`if`). Notar que no se pide que las dos ramas del `if` tengan el mismo tipo, esto es por el hecho de que en la mayoría de los programas, se usan condicionales precisamente para indicar

$\boxed{\Gamma \vdash P \blacktriangleright T}$

$$\langle \text{OUT} \rangle \frac{\Gamma \vdash u : \mathtt{ch}(\sigma) \qquad \Gamma \vdash e : \sigma \qquad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash u!\langle e \rangle; P \blacktriangleright \overline{u}; T}$$

$$\langle \text{IN} \rangle \frac{\Gamma \vdash u : \mathtt{ch}(\sigma) \qquad \Gamma, x : \sigma \vdash P \blacktriangleright T}{\Gamma \vdash u?(x); P \blacktriangleright u; T} \qquad \langle \text{TAU} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \tau; P \blacktriangleright \tau; T}$$

$$\langle \text{CLOSE} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \mathtt{close}\ \ u; P \blacktriangleright \mathtt{end}\ \ [u]; T} \qquad \langle \text{ZERO} \rangle \frac{}{\Gamma \vdash \mathbf{0} \blacktriangleright \mathbf{0}}$$

$$\langle \text{SEL} \rangle \frac{\Gamma \vdash \pi_i; P_i \blacktriangleright \kappa_i; T_i}{\Gamma \vdash \mathtt{select}\{\pi_i; P_i\}_{i \in I} \blacktriangleright \&\{T_i\}_{i \in I}}$$

$$\langle \text{IF} \rangle \frac{\Gamma \vdash e : \mathtt{bool} \qquad \Gamma \vdash P \blacktriangleright S \qquad \Gamma \vdash Q \blacktriangleright T}{\Gamma \vdash \mathtt{if}\ \ e\ \ \mathtt{then}\ \ P\ \ \mathtt{else}\ \ Q \blacktriangleright \oplus\{T_i\}_{i \in I}}$$

$$\langle \text{NEW} \rangle \frac{\Gamma, y : \mathtt{ch}(\sigma) \vdash P \blacktriangleright T \qquad c \notin \mathtt{dom}(\Gamma) \bigcup fn(T)}{\Gamma \vdash \mathtt{newchan}(y : \sigma); P \blacktriangleright (\mathtt{new}\ \ c)T\{c/y\}}$$

$$\langle \text{PAR} \rangle \frac{\Gamma \vdash P \blacktriangleright T \qquad \Gamma \vdash Q \blacktriangleright S}{\Gamma \vdash P \,|\, Q \blacktriangleright (T \,|\, S)} \qquad \langle \text{VAR} \rangle \frac{\Gamma \vdash \tilde{e} : \tilde{\sigma} \qquad \Gamma \vdash \tilde{u} : \mathtt{ch}(\tilde{\sigma}')}{\Gamma, X(\tilde{\sigma}, \mathtt{ch}(\tilde{\sigma}')) \vdash X\langle \tilde{e}, \tilde{u} \rangle \blacktriangleright \mathbf{t}_X \langle \tilde{u} \rangle}$$

$\boxed{\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}}$

$$\langle \text{DEF} \rangle \frac{\begin{array}{c}\Gamma, X_i(\tilde{\sigma}_i, \mathtt{ch}(\tilde{\sigma}_i')), \tilde{x}_i : \tilde{\sigma}_i, \tilde{y}_i : \mathtt{ch}(\tilde{\sigma}_i') \vdash P_i \blacktriangleright T_i \\ \Gamma, X_1(\tilde{\sigma}_1, \mathtt{ch}(\tilde{\sigma}_1')), \ldots, X_n(\tilde{\sigma}_n, \mathtt{ch}(\tilde{\sigma}_n')) \vdash Q \blacktriangleright S\end{array}}{\Gamma \vdash \{X_i(\tilde{x}_i, \tilde{y}_i) = P_i\}_{i \in I}\ \ \mathtt{in}\ \ Q \blacktriangleright \{\mathbf{t}_{X_i}(\tilde{y}_i = T_i)\}_{i \in I}\ \ \mathtt{in}\ \ S}$$

$\boxed{\Gamma \vdash_B P \blacktriangleright T}$

$$\langle \text{INT} \rangle \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash_\emptyset P \blacktriangleright T} \qquad \langle \text{RES} \rangle \frac{\Gamma, c : \mathtt{ch}(\sigma) \vdash_B P \blacktriangleright T}{\Gamma \vdash_{B \setminus c} (\nu c)P \blacktriangleright (\nu c)T}$$

$$\langle \text{CBUFF} \rangle \frac{\Gamma \vdash a : \mathtt{ch}(\sigma)}{\Gamma \vdash_{\{a\}} a^\star\langle \sigma \rangle :: \tilde{v} \blacktriangleright a^\star} \qquad \langle \text{BUFF} \rangle \frac{\Gamma \vdash a : \mathtt{ch}(\sigma)}{\Gamma \vdash_{\{a\}} a\langle \sigma \rangle :: \tilde{v} \blacktriangleright \lfloor a \rfloor}$$

$$\langle \text{PARR} \rangle \frac{\Gamma \vdash_B P \blacktriangleright T \qquad \Gamma \vdash_{B'} Q \blacktriangleright S \qquad B \bigcap B' = \emptyset}{\Gamma \vdash_{B \bigcup B'} P \,|\, Q \blacktriangleright (T \,|\, S)}$$

*Fig. 3.5:* Reglas de tipado (procesos y programas)

que va a haber diferentes comportamientos. Las reglas ⟨CLOSE⟩, ⟨ZERO⟩, ⟨PAR⟩ y ⟨TAU⟩ se deducen de su definición. La regla ⟨NEW⟩ estipula que la creación de un canal referenciado por la variable $y$ de tipo $\sigma$ seguida del proceso $P$, tiene tipo $(\texttt{new}\ c)T\{^{c}/_{y}\}$, donde $c$ (variable fresca) es el nuevo canal referenciado por $y$, y $T$ es el tipo comportamental de $P$. Por último, la regla ⟨VAR⟩ denota que la invocación a una función $X$ tiene tipo $\mathbf{t}_X\langle\tilde{u}\rangle$ siempre y cuando esté definida en el contexto con parámetros de tipo $\tilde{\sigma}$ y $\texttt{ch}(\tilde{\sigma'})$ (para expresiones y canales respectivamente) y que los argumentos tengan el tipo apropiado con respecto a ese contexto.

El juicio $\Gamma \vdash \mathbf{P} \blacktriangleright \mathbf{T}$, hace referencia a la regla de tipado de un programa. Un programa se define como un proceso principal ($Q$) asociado a un conjunto de definiciones de procesos ($P_i$), y va a tener tipo, siempre que todas las funciones tengan tipo, y el proceso principal también lo tenga al agregar esas definiciones al contexto.

Por último, en la figura 3.5 también están definidas las reglas para el juicio de tipado $\Gamma \vdash_B \mathbf{P} \blacktriangleright \mathbf{T}$, y tipan dinámicamente a un proceso creado después de la ejecución de un programa: en el contexto $\Gamma$, el proceso $P$ tendrá el tipo comportamental $T$ con un conjunto $B$ de *buffers* asociados a los canales y creados en tiempo de ejecución. Cuando se quiere ligar un canal en tiempo de ejecución se utiliza la regla de tipado ⟨RES⟩, donde dado un proceso de tipo $T$ que puede usar el canal $c$, se tipa a $(\nu c)P$ con $(\nu c)T$ sacando a $c$ del conjunto $B$ ya que es local a $P$ (y a $T$). Las reglas ⟨CBUFF⟩ y ⟨BUFF⟩ establecen que los *buffers* abiertos ($a\langle\sigma\rangle$) y cerrados ($a^{\star}\langle\sigma\rangle$) son de tipo $\lfloor a \rfloor$ y $a^{\star}$ respecticamente si el canal al que están asociados es del tipo correspondiente ($\texttt{ch}(\sigma)$) y el *buffer* $a$ fue definido previamente en la ejecución ($\vdash_a$). Por último, la regla ⟨PARR⟩ garantiza que los *buffers* de ambos procesos no se superpongan (por lo tanto, solo existe un único *buffer* para cada nombre en el contexto).

## 3.3. Tipos como procesos

Una vez definida la sintaxis para los tipos, procedemos a definir la semántica simbólica, es decir, aquello que va a determinar el comportamiento. Esto genera un *Labelled Transition System* (LTS) que va a tener una cantidad de estados finita (siempre y cuando los tipos validen la propiedad *fenced*) y que, por lo tanto, puede ser ejecutado simbólicamente.

En la figura 3.6 están dadas las reglas de semántica operacional. Las etiquetas $\alpha$ y $\beta$ son de la forma:

$$\alpha\,,\,\beta \;\;:=\;\; \overline{a} \;\mid\; a \;\mid\; \tau \;\mid\; [a] \;\mid\; \texttt{end}[a] \;\mid\; \overline{\texttt{end}}[a] \;\mid\; a^{\star}$$

y denotan acciones de envío y recepción, transiciones silenciosas, sincronización sobre un canal, el pedido y aprobación del cierre de un canal, y acciones de envío del valor por defecto de un canal cerrado, respectivamente.

Las reglas |SND| y |RCV| permiten a un tipo emitir una acción de enviar o recibir sobre un canal $a$. La regla |TAU| permite continuar la ejecución luego de una acción silenciosa. |SEL| modela la elección no determinística, y denota que el tipo puede reducirse a cualquiera de las acciones contenidas en la elección a través de una transición silenciosa. |BRA| se refiere a las elecciones internas, y estipula que se puede reducir a $T_j$ mediante una acción $\alpha$ siempre que el término $\kappa_j; T_j$ contenido en la elección reduzca a $T_j$ mediante la acción $\alpha$. La regla |COM| permite a dos tipos, que se están ejecutando en paralelo, sincronizar a través de acciones duales: enviar y recibir, o recibir de un canal cerrado. La regla |NEW| denota que desde un estado donde se crea un canal $a$ previo al tipo $T$, se transiciona mediante $\tau$ a una composición en paralelo de $T$ con un *buffer* abierto, y en ambos se liga el canal con la variable $a$.

Las reglas |END|, |BUF| y |CLOSE| modelan el cierre de un canal: la primera establece que para un tipo compuesto por $\texttt{end}[a]$ seguido de $T$, se transiciona a $T$ mediante la acción $\texttt{end}[a]$; la segunda transiciona a través de $\overline{\texttt{end}}[a]$ desde un *buffer* abierto a uno cerrado; la tercera unifica estas dos reglas para sincronizar el cierre de un canal entre dos procesos en paralelo (el que lo solicita, y el que se cierra). La regla |CLD| modela la capacidad de un *buffer* cerrado de siempre enviar el valor por defecto de su tipo.

La regla |RES-1| establece que $(\nu a)T$ reduce a $(\nu a)T'$ con una transición $\alpha$ si $T$ reduce a $T'$ a través de $\alpha$ y $\alpha$ no es una acción sobre el canal $a$. En cambio si la reducción de $T$ a $T'$ es por una sincronización sobre el canal $a$, la regla |RES-2| establece que la transición es $\tau$. Estas dos reglas son las que modelan el comportamiento general de la ejecución y que respetan la semántica de Go, mediante la cual no se puede reducir la acción de enviar o recibir por un canal si en paralelo no existe otro proceso que recibe o envía. Por último, la regla |DEF| establece que el tipo de la invocación a una función reduce a lo que reduciría el tipo del programa referenciado por esa invocación, reemplazando los parámetros correspondientes.

En la figura 3.7 se define la congruencia estructural para los tipos. Se detallan reglas de asociatividad, conmutatividad, eliminación de operadores, etc. La ejecución simbólica de los tipos está definida por las reglas de semántica operacional módulo congruencia estructural. Escribimos $\rightarrow$ para $\xrightarrow{\tau} \cup \equiv$ y $T \rightarrow^\star \xrightarrow{\alpha}$ si existen $T'$ y $T''$ tal que $T \rightarrow^\star T' \xrightarrow{\alpha} T''$.

$$|\text{SND}|\ \ \overline{a};T \xrightarrow{\overline{a}} T \qquad |\text{RCV}|\ \ a;T \xrightarrow{a} T \qquad |\text{TAU}|\ \ \tau;T \xrightarrow{\tau} T$$

$$|\text{SEL}|\ \frac{j \in I}{\oplus\{T_i\}_{i\in I} \xrightarrow{\tau} T_j} \qquad |\text{BRA}|\ \frac{\kappa_j;T_j \xrightarrow{\alpha} T_j}{\&\{\kappa_i;T_i\}_{i\in I} \xrightarrow{\alpha} T_j} \qquad |\text{PAR}|\ \frac{T \xrightarrow{\alpha} T'}{T \mid S \xrightarrow{\alpha} T' \mid S}$$

$$|\text{COM}|\ \frac{T \xrightarrow{\beta} T' \qquad S \xrightarrow{a} S' \qquad \beta = \overline{a}, a^\star}{T \mid S \xrightarrow{[a]} T' \mid S'} \qquad |\text{NEW}|\ \ (\texttt{new}\ \ a)T \xrightarrow{\tau} (\nu a)(T \mid \lfloor a \rfloor)$$

$$|\text{END}|\ \ \texttt{end}[a];T \xrightarrow{\texttt{end}[a]} T \qquad |\text{BUF}|\ \ \lfloor a \rfloor \xrightarrow{\overline{\texttt{end}}[a]} a^\star \qquad |\text{CLD}|\ \ a^\star \xrightarrow{a^\star} a^\star$$

$$|\text{CLOSE}|\ \frac{T \xrightarrow{\texttt{end}[a]} T' \qquad S \xrightarrow{\overline{\texttt{end}}[a]} S'}{T \mid S \xrightarrow{\tau} T' \mid S'} \qquad |\text{EQ}|\ \frac{T \equiv_\alpha T' \qquad T \xrightarrow{\alpha} T''}{T \xrightarrow{\alpha} T''}$$

$$|\text{RES-1}|\ \frac{T \xrightarrow{\alpha} T' \qquad fn(\alpha) \neq \{a\}}{(\nu a)T \xrightarrow{\alpha} (\nu a)T'} \qquad |\text{RES-2}|\ \frac{T \xrightarrow{[a]} T'}{(\nu a)T \xrightarrow{\tau} (\nu a)T'}$$

$$|\text{DEF}|\ \frac{T\{\tilde{a}/\tilde{x}\} \xrightarrow{\alpha} T' \qquad \mathbf{t}(\tilde{x}) = T}{\mathbf{t}\langle\tilde{a}\rangle \xrightarrow{\alpha} T'}$$

*Fig. 3.6:* Semántica operacional de tipos

$$T \mid S \equiv S \mid T \qquad T \mid (T' \mid S) \equiv (T \mid T') \mid S \qquad T \mid \mathbf{0} \equiv T$$
$$(\nu a)(\nu b)T \equiv (\nu b)(\nu a)T \quad (\nu a)\mathbf{0} \equiv \mathbf{0} \quad (\nu a)a^\star \equiv \mathbf{0} \quad (\nu a)\lfloor a \rfloor \equiv \mathbf{0}$$
$$T \mid (\nu a)S \equiv (\nu a)(T \mid S)\ \ (a \notin fn(T)) \qquad T \equiv_\alpha T' \Rightarrow T \equiv T'$$

*Fig. 3.7:* Congruencia estructural de tipos

## 3.4. Barbas

En [3] se definen las barbas como predicados atómicos aplicados a procesos y que representan la existencia o no de una acción de comunicación. Estos predicados se crean para poder definir formalmente las propiedades *liveness* y *safety*. Se escriben de la forma $P \downarrow_o$ en donde $P$ es un proceso y $o \in \{a, \overline{a}, [a], \texttt{end}[a], a^\star\}$ de manera que: $P \downarrow_a$ denota que el proceso $P$ está listo para realizar una acción de recibir de un canal $a$; análogamente $P \downarrow_{\overline{a}}$ denota que $P$ está listo para enviar; $P \downarrow_{[a]}$ indica que $P$ puede realizar una sincronización sobre $a$; $P \downarrow_{\texttt{end}[a]}$ indica que $P$ puede cerrar el canal $a$; y $P \downarrow_{a\star}$ indica que $P$ puede mandar el valor por *default* del canal cerrado $a$. En la figura 3.8 podemos ver las definiciones de las barbas para la semántica operacional de la figura 3.6:

$$c?(x) \downarrow_c \qquad c!\langle e\rangle \downarrow_{\overline{c}} \qquad \texttt{close}\ c; Q \downarrow_{\texttt{end}[c]} \qquad c^\star\langle\sigma\rangle :: \tilde{v} \downarrow_{c^\star}$$

$$\frac{P \downarrow_o}{P \mid Q \downarrow_o} \qquad \frac{\pi \downarrow_o}{\pi; Q \downarrow_o} \qquad \frac{P \downarrow_o \quad a \notin fn(o)}{(\nu a)P \downarrow_o} \qquad \frac{P \downarrow_o \quad P \equiv Q}{Q \downarrow_o}$$

$$\frac{Q\{^{\tilde{c},\tilde{a}}/_{\tilde{x}}\} \downarrow_o \qquad X(\tilde{x} = Q)}{X\langle \tilde{e}, \tilde{a}\rangle \downarrow_o} \qquad \frac{\forall i \in \{1, ..., n\} : \pi_i \downarrow_{o_i}}{\texttt{select}\{\pi_i; P_i\}_{i\in\{1,...,n\}} \downarrow_{\{o_1 ... o_n\}}}$$

$$\frac{P \downarrow_a \qquad Q \downarrow_{\overline{a}} \ \texttt{or}\ Q \downarrow_{a^\star}}{P \mid Q \downarrow_{[a]}} \qquad \frac{P \downarrow_a \qquad \pi_i \downarrow_{\overline{a}}}{P \mid \texttt{select}\{\pi_i; P_i\}_{i\in I} \downarrow_{[a]}} \qquad \frac{P \downarrow_{\overline{a}} \ \texttt{or}\ P \downarrow_{a^\star} \qquad \pi_i \downarrow_a}{P \mid \texttt{select}\{\pi_i; P_i\}_{i\in I} \downarrow_{[a]}}$$

$$P \Downarrow_o \text{ si } P \rightarrow^\star P' \text{ y } P' \downarrow_o.$$

*Fig. 3.8:* Predicados de las barbas

Podemos observar que si un proceso (o prefijo $\pi$) satisface cierto predicado $\downarrow_o$, la composición en paralelo (o secuencial) con cualquier otro proceso también lo hará. Lo mismo sucede para el caso $(\nu a)P$, siempre y cuando el canal que se liga $(a)$ no forme parte de los nombres libres en $o$. Dos procesos que son equivalentes, satisfacen las mismas barbas. La invocación a una función va a satisfacer los predicados de la función invocada, siempre y cuando se reemplacen bien las variables correspondientes que se pasan por parámetro. Un `select` tiene un conjunto de barbas $\tilde{o}$ que se corresponden con las barbas de cada una de las ramas que puede ejecutar. Finalmente, una composición en paralelo va a satisfacer el predicado de sincronización ($\downarrow_{[a]}$) cuando ocurra alguna de estas tres opciones: un proceso recibe y otro manda un valor sobre el mismo canal (puede ser el valor por defecto de un canal cerrado); un proceso recibe y en paralelo hay un `select` con una de sus ramas que envía sobre el mismo canal; un proceso envía (puede ser el valor por defecto) y en paralelo hay un `select` con una de sus ramas que recibe sobre el mismo canal.

## 3.5. Liveness y Safety

Una vez definidos los predicados de comunicación (barbas) estamos en condiciones de definir formalmente cuándo un programa cumple con las propiedades *liveness* y *safety*.

Un programa es ***live*** si, para todos los estados alcanzables del LTS generado, se cumple que: si el estado puede generar una acción de envío o recepción sobre un canal, eventualmente podrá realizar una sincronización sobre el mismo canal; y si el estado puede realizar un conjunto de acciones (por ejemplo un `select`), eventualmente podrá realizar una sincronización con alguna de ellas.

O formalmente, el programa $\mathbf{P}$ satisface *liveness* si $\forall Q$ tal que $\mathbf{P} \rightarrow^{\star} (\nu\tilde{c})Q$ vale que:

a) Si $Q \downarrow_{a}$ o $Q \downarrow_{\overline{a}}$ entonces $Q \Downarrow_{[a]}$; y

b) Si $Q \downarrow_{\overline{a}}$ entonces $Q \Downarrow_{[a_i]}$ para algún $a_i \in \{\tilde{a}\}$.

Un programa es ***safe*** si en todos los estados alcanzables, no se intenta cerrar un canal cerrado y ningún proceso realiza una acción de envío sobre un canal cerrado. Formalmente, el programa $\mathbf{P}$ satisface *safety* si:

$$\forall Q \text{ tal que } \mathbf{P} \rightarrow^{\star} (\nu\tilde{c})Q \text{ si } Q \downarrow_{a^{\star}} \text{ entonces } \neg(Q \Downarrow_{\mathtt{end}[a]}) \text{ y } \neg(Q \Downarrow_{\overline{a}})$$

## 3.6. Implementación

La implementación del análisis estático para programas en Go consta de una cadena de herramientas de verificación que se puede ver en la figura 3.9: la primera herramienta analiza el código fuente, lo parsea a una representación SSA intermedia y luego a código MiGo; la segunda toma el código MiGo, lo parsea a una representación intermedia, construye su tipo comportamental y finalmente chequea si cumple las propiedades *liveness* y *safety*.

### 3.6.1. Dingo Hunter

Dingo Hunter es una herramienta que consta de tres etapas: la primera toma el código Go que se quiere analizar y lo parsea a una representación intermedia SSA; en la segunda se abstraen las acciones de comunicación que interesan y se transforman en estructuras llamadas *statements*; y finalmente en la tercera se extrae el código MiGo de los *statements*.

El código en MiGo devuelto tiene una sintaxis que difiere un poco de la definida en la figura 3.1. Esto sucede no solo porque la herramienta filtra las instrucciones quedándose

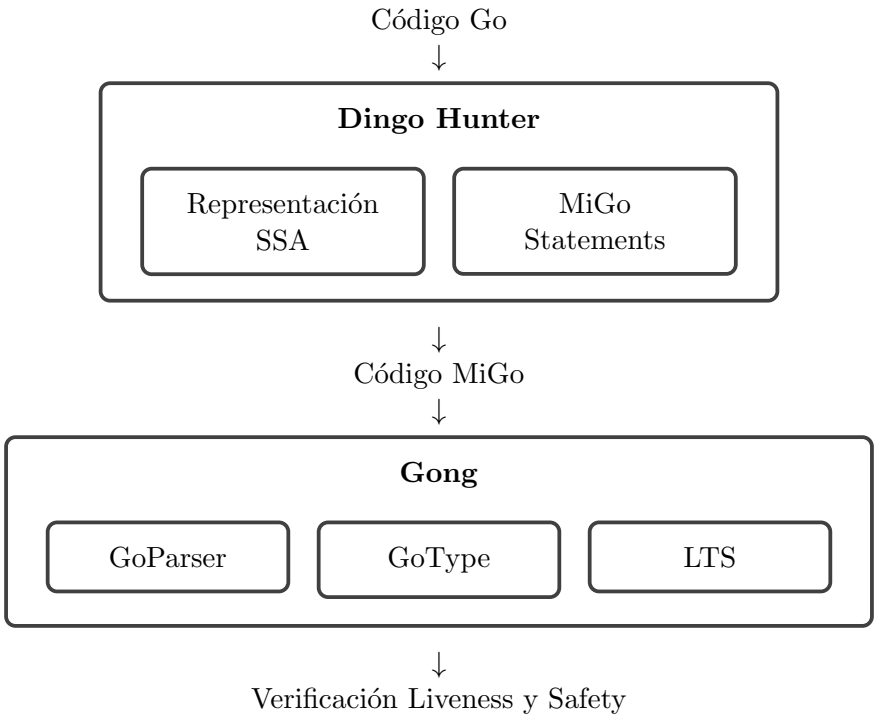


*Fig. 3.9:* Workflow de la cadena de verificación

con las que corresponden a acciones de comunicación, si no que además se modifican algunos operadores y se eliminan parámetros que no son relevantes. Se puede observar en la figura 3.10 el código MiGo que genera la herramienta para el programa que calcula el factorial de la figura 2.1

Por otro lado, en [6] se le incorpora a algunas instrucciones el símbolo @ seguido por un número entero, que hace referencia al número de línea de esa instrucción en el programa original de Go. Este cambio se ve motivado por la necesidad de agregar información de la historia de ejecución de un programa. Es por esto que las instrucciones a las que se les agrega su número de línea son `send`, `recv`, `tau`, `close`, `spawn`[1], `call` y `newchan`, ya que se considera que aportan información relevante para dicho fin.

### 3.6.2. Gong

Gong es una herramienta desarrollada en Haskell, que tiene como valor de *input* un programa escrito en MiGo y devuelve *Strings* en donde se detalla si se verifican *safety* y *liveness* (la extensión propuesta en [6] agrega información sobre qué instrucciones violan esas propiedades). El funcionamiento de esta herramienta consiste en varias etapas: pri-

[1] `spawn` representa la invocación a una función que se va a ejecutar en otra go-rutina, es decir, a la ejecución en paralelo.

```
def main():
    let t0 = newchan main.t0_0_0, 0 @2;
    spawn fact(t0) @3;
    recv t0 @4;
def fact(b):
    if send b @8;
    else let t1 = newchan fact.t1_0_0, 0 @11;
        spawn fact(t1) @12;
        recv t1 @13;
        send b @13;
    endif;
```

*Fig. 3.10:* Código MiGo obtenido luego de ejecutar Dingo Hunter sobre el programa de la figura 2.1

mero se parsea el programa en MiGo y se obtiene una representación intermedia para los *statements*, realizando una validación sintáctica del programa; luego se infiere el tipo comportamental (**GoType**) y se verifica que sea *fenced*; a continuación se genera el LTS que representa la ejecución simbólica del programa; finalmente se recorren todos los estados de la ejecución simbólica para verificar las propiedades deseadas.

#### 3.6.2.1. GoParser

En el archivo `GoParser.hs` se define un *parser* que transforma el código MiGo, primero en una representación intermedia que podemos observar en la figura 3.11, y luego en un **GoType**, de lo que hablaremos en la siguiente sección.

La función encargada de esta transformación se llama `parseprog` y toma un argumento de tipo `String` que representa el código escrito en MiGo y devuelve el programa en la representación intermedia llamado `ProgGo`. La idea consiste en mapear las instrucciones de MiGo y colocarlas en una lista (`Seq [Interm]`), de manera de mantener el orden en la cual se ejecutan. Una instancia de `ProgGo` será un programa que consta de una lista de definiciones de funciones de tipo `Interm`.

A los constructores que se corresponden con las operaciones `call`, `close`, `spawn`, `newchan`, `tau`, `send` y `recv`, se les incorporó un parámetro de tipo `Int` que representa el número de línea de esa instrucción en el programa original.

#### 3.6.2.2. GoType

El tipo comportamental sobre el cual se va a trabajar para verificar las propiedades se llama `GoType`, está definido en el archivo `GoTypes.hs` y podemos ver su representación en la figura 3.12.

```
data Prog a = P [Def a]  deriving (Eq, Show)

data Def a =  D String [String] a  deriving (Eq, Show)

type ProgGo = Prog Interm

data Interm = Seq [Interm]
            | Call Int String [String]
            | Cl Int String
            | Spawn Int String [String]
            | NewChan Int String String Integer
            | If Interm Interm
            | Select [Interm]
            | T Int
            | S Int String
            | R Int String
            | Zero
        deriving (Eq, Show)
```

*Fig. 3.11:* Representación intermedia

Para representar un programa en MiGo se define el tipo `Eqn`, que contiene un constructor `EqnSys`, una lista de funciones auxiliares, y el programa principal (se corresponde con la regla ⟨DEF⟩ de la figura 3.5). Las funciones auxiliares se definen como una tupla `(EqnName, Embed GoType)` con el nombre de la función y el tipo que describe su comportamiento. El nombre se representa con el tipo `EqnName`, que es un renombre del tipo `Name a` de *Haskell*. Éste representa nombres de elementos de tipo `a` que se pueden ligar (en este caso de `GoTypes`). La mónada `Bind`[12] representa que los elementos del primer argumento (lista de funciones) se ligan en el segundo (programa principal).

Luego se define el tipo `GoType` que representa a la sintaxis descripta en la figura 3.5. Todos los argumentos de tipo `String` en los constructores representan la historia de ejecución definida en la extensión [6] al momento de ejecutar esa instrucción. Hablaremos de ello más adelante. Los constructores `Send` y `Recv` tienen como argumento el nombre del canal y el tipo que se ejecutará a continuación (lo mismo sucede con `tau` pero sin el canal). `IChoice` representa las elecciones internas, sus argumentos son los dos posibles tipos que se pueden elegir. `OChoice` por su parte representa a la elección no determinística, y en vez de tener dos argumentos, tiene uno solo que es una lista de tipos.

La composición paralela se representa con el constructor `Par`, que tiene como argumento una lista de tipos que se ejecutan en paralelo. `New` representa la creación de un

```
data Eqn = EqnSys (Bind (Rec [(EqnName, Embed GoType)]) GoType) deriving (Show)

type EqnName = Name GoType

data GoType = Send String ChName GoType
            | Recv String ChName GoType
            | Tau String GoType
            | IChoice String GoType GoType
            | OChoice String [GoType]
            | Par String [GoType]
            | New String Int (Bind ChName GoType)
            | Null
            | Close String ChName GoType
            | TVar String EqnName
            | ChanInst GoType [ChName]
            | ChanAbst (Bind [ChName] GoType)
            | Seq String [GoType]
            | Buffer ChName (Bool, Int, Int)
            | ClosedBuffer ChName
    deriving (Show)

type ChName = Name Channel
```

*Fig. 3.12:* Tipo Comportamental GoType

canal: tiene como argumento un entero que representa el tamaño del *buffer* y un tipo que se ejecutará a continuación, al cual se le liga el nombre del canal creado utilizando la mónada `Bind`. Para finalizar la ejecución se utiliza el tipo `Null`.

El constructor `Close` representa el cierre de un canal, es por esto que sus argumentos son el nombre del canal y el tipo que se ejecuta a continuación. La regla ⟨VAR⟩ que representa la invocación de funciones, se corresponde con dos constructures: `ChanInst` que representa el pasaje como parámetros de una lista de canales a una función invocada en el primer argumento, que solo puede ser de tipo `TVar`, que tiene como argumento el nombre de la función a invocar.

`ChanAbst` representa la definición de una función, que tiene como argumentos una lista de canales que se ligan a un `GoType`. Este tipo se utiliza para definir a las funciones auxiliares dentro del contexto del `EqnSys`. `Seq` representa la ejecución secuencial de la lista de tipos que toma como argumento, se utiliza únicamente para modelar la invocación a una función seguida de otro tipo, y solo acepta listas de un elemento, o de dos, en cuyo caso el segundo elemento es el tipo `Null`. `ClosedBuffer` representa a un canal cerrado cuyo nombre está en el primer argumento. Finalmente, el constructor `Buffer` representa a un *buffer* de un determinado canal y tiene información acerca de si está abierto o cerrado, su capacidad y la cantidad de elementos que tiene.

El parámetro de tipo *String* surge de la extensión en [6], y no está presente en todos los constructores. Esto es por dos motivos: el primero es que el constructor no se corresponde a una instrucción real del programa en Go o que la información sobre el número de línea no resulta relevante, es el caso de `Null`, `Buffer`, `ClosedBuffer` y `ChanAbst`; el segundo es porque es un constructor que se representa mediante la composición de dos `GoTypes` distintos, por lo que se decidió agregar el campo solo en uno de ellos (es el caso de `ChanInst` y `TVar`).

El parseo completo se realiza dentro del archivo `GoParser.hs`, `fullPass` es la función más general encargada de transformar un texto de un programa en MiGo a un `GoType`. Dentro de esta, la función que toma una representación intermedia (`ProgGo`) y devuelve el tipo comportamental `Eqn` es `TransformProg`: primero separa la lista de definiciones y se queda con el primer elemento que será el programa principal, y el resto serán las definiciones de las funciones auxiliares; a cada una le aplica una función de transformación diferente y luego forma el tipo `Eqn`.

Las funciones encargadas de parsear las definiciones y el programa principal intermedios son `transformDef` y `transformMain` respectivamente, pero ambas terminan invocando a `transformSeq` para parsear cada `Interm` dentro de la secuencia. Es esta última

función la encargada de hacer *pattern matching* entre los constructores del tipo `Interm` y los del `GoType`. `transformSeq` toma además como argumento una lista de `String` llamado `vars` que representa al contexto, se inicializa vacío, y cada vez que se crea un canal con el constructor `NewChan`, se agrega el nombre del mismo a la lista.

`transformSeq` también inicializa el valor de tipo *String* que representa la historia de ejecución de cada `GoType`. Para las instrucciones `IChoice`, `OChoice` y `New` no hay información relevante que agregar al constructor y se genera con el *String* vacío, el cuál se utilizará para acumular información y pasarla a instrucciones siguientes. Para las instrucciones `Send`, `Recv`, `Tau` y `Close` se añade información del número de línea. Y para las instrucciones `Par` y `Seq`, como se generan al hacer una invocación a una función, se agrega la información de la línea al constructor `TVar` que va dentro de la lista de `GoTypes`.

En la figura 3.13 se puede observar el tipo comportamental generado para el programa de la figura 2.1. En el contexto se define la función `fact` mediante el constructor `ChanAbst` que liga el canal `b` al `GoType` que representa la función. Éste se define como una elección interna `IChoice` cuya primera rama envía sobre `b`, y la segunda un nuevo canal `b2` y ejecuta en paralelo las acciones de: invocar a `fact` con `b2` como argumento, y luego recibir sobre `b2` y enviar sobre `b`. Luego está el `GoType` del método principal, que crea un canal `a`, y lanza en paralelo la invocación a `fact` y la recepción sobre el canal.

```
EqnSys (bind [[(fact,{ChanAbst (bind [b] (IChoice "" (Send "4" 0@0 Null)
(New "5" 0 (bind b2 (Par "6" [ChanInst (TVar "SPAWN on Line 6" 2@0) [0@0],
Recv "7" 0@0 (Send "8" 1@0 Null)])))))})]] (New "1" 0 (bind a (Par "2"
[ChanInst (TVar "SPAWN on Line 2" 1@0) [0@0],Recv "3" 0@0 Null]))))
```

*Fig. 3.13:* `GoType` generado para el programa de la figura 2.1

#### 3.6.2.3. Labeled Transition System

La lógica de la construcción de la máquina de estados que representa la ejecución simbólica de los tipos comportamentales se encuentra en el archivo `SymbolicSem.hs`, y es la función `succs` la encargada de esta transformación: toma como argumento un entero que representa la cota de iteraciones (se calcula teniendo en cuenta el tamaño del tipo y de las llamadas recursivas dentro de este), y el programa de tipo `Eqn`, y devuelve una lista de `Eqn` que representan los estados del LTS.

Los estados del LTS son tipos que representan un momento en la ejecución que necesita de una acción de sincronización para avanzar, y las aristas representan esa acción que, una vez ejecutada, lleva al estado siguiente. Esto nos permite realizar la ejecución

simbólica del programa y verificar propiedades para cada estado. Si en algún momento se quiere verificar un estado que ya fue explorado, entonces no será necesario hacerlo; esto garantiza que se puedan analizar programas infinitos.

Para generar los estados es necesario definir cuáles son las instrucciones que generan **guardas**. Este concepto se utiliza para definir qué tipos necesitan de una acción de sincronización en algún proceso ejecutándose en paralelo para poder progresar. Se subclasifican en guardas *bloqueantes* o guardas *tau*.

Las guardas *bloqueantes* están asociadas a las acciones que necesitan de otra para sincronizar. Pueden ser de dos tipos:

- de comunicación: se corresponden con las acciones que se comunican a través de un canal. Pueden ser `Send`, `Recv` o una elección no determinística `OChoice`, ya que esta puede ejecutar cualquiera de sus ramas dependiendo de si alguna de sus guardas sincroniza;
- de cierre de un canal: si bien en un programa en Go solo existe la instrucción `close` para cerrar un canal, internamente el compilador involucra un hilo de ejecución que realiza el pedido de cierre y otro que lo cierra. Las acciones asociadas a estas guardas son `Close` y `Buffer`.

Las guardas *tau* están asociadas a la acción silenciosa de tipo $\tau$. Esta acción no requiere que existan procesos ejecutándose en paralelo, sino que puede realizar la sincronización por sí misma.

La función que genera la lista de guardas para cada acción se llama `getGuardsCont`. Cada guarda se representa con una tupla cuyo primer elemento es la acción de sincronización, y el segundo es el tipo que se ejecuta a continuación. Es interesante notar que a la elección determinística `IChoice` se le asignan dos guardas silenciosas `tau`. Esto es porque para la ejecución simbólica se asume que ambas ramas del `if` van a ejecutarse independientemente de cuál sea la condición. Otras guardas interesantes son las que se crean para el tipo `Buffer`. En este caso, las acciones de sincronización dependen de los parámetros del *buffer*: si está abierto o cerrado, cuál es su capacidad y cuántos elementos contiene.

La función `succs` utiliza la función `genStates` para generar los estados del LTS. Ésta recibe 5 argumentos: un entero indicando la cota, una lista con nombres de canales, un contexto en donde se encuentran las definiciones de las funciones auxiliares, una lista de estados ya recorridos (inicialmente la lista vacía) y una lista de estados por recorrer (inicialmente con el primer estado como único elemento). La función itera sobre esta última

lista. Cada vez que recorre un estado, lo saca de la lista de estados por recorrer y lo agrega a la de ya recorridos. Cuando se genera un nuevo estado, si no está en la lista de ya recorridos, se agrega a la de estados por recorrer, y se llama recursivamente para ver si se pueden generar más estados a partir de este. La manera en que se calcula el estado siguiente es a través de la composición de dos funciones: `genSuccs` y `normalise`.

`genSuccs` es la encargada de devolver una lista de estados siguientes calculados a partir de la compatibilidad entre las guardas. La idea consiste en generar una lista de tipos que se ejecutan en paralelo y para cada uno de ellos generar las guardas. Éstas se recorren y una vez que se identifica cuáles son compatibles entre sí, es decir, cuáles pueden sincronizar, se genera un estado nuevo reemplazando la instrucción que generó la guarda por el segundo elemento de la tupla, que representaba el tipo que se ejecutaba a continuación.

A cada uno de los estados generados por `genSuccs` se le aplica luego la función `normalise`. Ésta es la encargada de aplicar las reglas de congruencia estructural. Primero invoca a `unfoldTillGuard`, que devuelve el tipo de una composición en paralelo con todos los tipos que están listos para sincronizar dentro del estado. Para realizar esto, `unfoldTillGuard` recorre las estructuras de los tipos que no generan guardas (`Par`, `ChanInst (TVar)`, `New`, `ChanAbst` y `Seq`), "ejecuta" las acciones correspondientes asociadas al tipo (crear canales y reemplazar invocaciones a funciones), y se llama recursivamente con el tipo que continúa. Los tipos que generan guardas se devuelven sin ninguna modificación. La clave de esta función es que detiene la ejecución si supera a la cota que se le pasa como parámetro o si recorre por segunda vez una función que ya fue invocada en la misma ejecución. Además, tiene un parámetro de tipo *String* llamado *trace* que se utiliza para pasar información sobre la historia de la ejecución previa a los tipos subsiguientes. Cada llamada recursiva utiliza el valor del *trace* anterior y le agrega información sobre la nueva llamada. Una vez que `unfoldTillGuard` devuelve el estado, `normalise` inicializa los canales y sus *buffers* asociados aplicando las reglas de congruencia estructural. Ésto se realiza componiendo las funciones `initiate`, `gcBuffer` y `nf`.

En la figura 4.4 se puede observar el LTS construido para el `GoType` de la figura 3.13. Se procedió a eliminar ciertos constructores y cambiar la notación para facilitar la legibilidad y comprensión del mismo. Cada estado consta de una lista que representa los tipos que se ejecutan en paralelo.

Para construir el estado inicial, se reemplaza la invocación a la función `fact` por su definición correspondiente, se mantiene la acción de recibir sobre el canal `a`, y se inicializa el *buffer* del mismo. Como nos encontramos frente a una elección interna, y el generador de LTS siempre asume que se van a ejecutar ambas ramas, se generan dos posibles transiciones para el estado 0. En la primera se reemplaza el `IChoice` por el tipo del primer

```
-- Estado 0:
[IChoice (Send a) (ChanInst fact(b), Recv b (Send a), Buffer b)), Recv a,
Buffer a]

-- Estado 1:
[Send a, Recv a, Buffer a]

-- Estado 2:
[Buffer a]

-- Estado 3:
[IChoice (Send b) (ChanInst fact(b2), Recv b2 (Send b), Buffer b2)),
Recv b (Send a), Buffer b, Recv a, Buffer a]

-- Estado 4:
[Send b, Recv b (Send a), Buffer b, Recv a, Buffer a]

-- Estado 5:
[ChanInst fact(b2), Recv b2 (Send b), Buffer b2, Recv b (Send a), Buffer b,
Recv a, Buffer a]
```

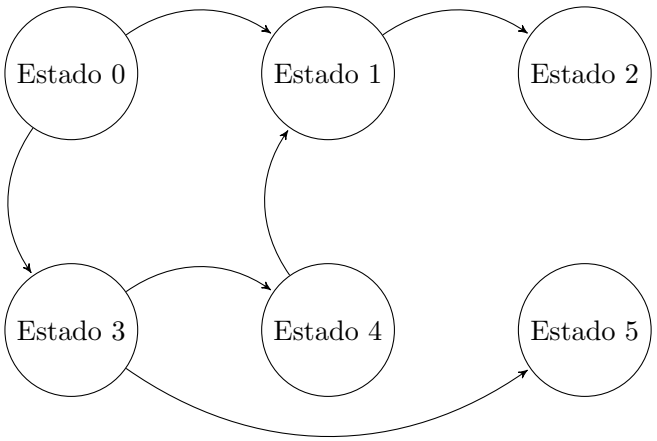


*Fig. 3.14:* LTS generado para el `GoType` de la figura 4.3

argumento, y se transiciona hacia le estado 1, que puede sincronizar sobre `a` (`Send-Recv`) y avanzar hacia el estado 2, que solo tiene información sobre su *buffer*, y es un estado final.

La otra alternativa en el estado 0 es reemplazar el `IChoice` por el segundo argumento, con lo que se transiciona al estado 3, pero reemplazando además la invocación `ChanInst` con la definición de `fact`. Es importante destacar que el nuevo tipo ahora utiliza al canal `b` en la primera rama, y crea un tercer canal `b2` para la llamada recursiva. En este punto existen de nuevo dos opciones de transición producidas por el `IChoice`. La primera es reemplazar la primera rama y avanzar hacia el estado 4, en donde se sincroniza sobre `b` y se vuelve al estado 1. La segunda opción es ejecutar la segunda rama del `IChoice`. Observemos que si se vuelve a hacer un reemplazo de la invocación a `fact`, se repite lo mismo que sucedía en la transición del estado 0 al 3, pero creando un nuevo canal. Esto se podría repetir hasta el infinito, y por este motivo es que existe la cota de iteraciones en `unfoldTillGuard`.

La cota que calcula Gong para este tipo es 2, y representa la máxima cantidad de invocaciones recursivas que va a realizar. Notar que las dos invocaciones ya fueron realizadas, la primera al momento de generar el estado inicial, y la segunda en la transición del estado 0 al 1. Por lo tanto, al haber alcanzado la cota, la invocación a `fact` no se realiza, y la segunda rama del `IChoice` se agrega sin modificaciones en el estado 5. Luego, éste no puede realizar ninguna acción de sincronización, y será un estado final.

#### 3.6.2.4. Verificación de Liveness

La verificación de esta propiedad la realiza la función `liveness` que está implementada en el archivo `Liveness.hs`, y toma como argumento una lista de `Eqn` que representa los estados del LTS generado a partir del programa que se quiere verificar. La idea general de la función consiste en recorrer los estados y para cada uno chequear que exista una acción de sincronización posible entre sus procesos que corren en paralelo y los de los de otros estados.

Para cada estado, primero se obtiene la lista de tipos que corren en paralelo. Esto se realiza a través de la función `extractType`, que devuelve la lista de tipos compuestos en paralelo si el estado es de tipo `Par`, o una lista con un único elemento si no (se ignora la creación de canales del tipo `New`).

Luego se recorren los tipos, y para cada uno se genera un nuevo LTS a partir de la composición en paralelo de todos los tipos excluyendo al tipo actual, es decir, se obtiene una lista de estados alcanzables mediante acciones de sincronización que no utilizan al tipo que se está recorriendo. Luego se generan las barbas del tipo actual (basado en las

definiciones de la figura 3.8) y se comparan con las barbas de los estados generados de manera de verificar que exista una posible sincronización. Este chequeo es lo que valida la propiedad *liveness*: en algún momento futuro de la ejecución se va a realizar una sincronización que le permita al programa progresar a partir del estado actual.

Si se encuentran estados que no tienen acciones con las cuales sincronizar, el programa no satisface *liveness* y la función devuelve un `String` especificando cuáles son esas operaciones y en qué línea del código original se encuentran.

#### 3.6.2.5. Verificación de Safety

La verificación de esta propiedad la realiza la función `safety` que está implementada en el archivo `Safety.hs`, y toma como argumento una lista de `Eqn` que representa los estados del LTS generado a partir del programa que se quiere verificar. La idea general de la función consiste en recorrer los estados verificando que los canales no se cierren dos veces y que no se intente enviar sobre un canal cerrado.

El procedimiento es similar al de `liveness`. Se recorre la lista de tipos que se componen en paralelo en cada estado. Si encuentra una instrucción `Close`, calcula un nuevo LTS a partir de la composición en paralelo de todos los estados pero reemplazando esa instrucción por un `Buffer` cerrado de ese canal, y agregando el tipo que se ejecutaba a continuación del `Close`.

Luego se calculan las barbas del tipo `Close` consumido y se compara con las de cada estado del LTS generado, verificando que no se realice una operación `Send` o `Close` sobre el mismo canal. Si alguna de las operaciones rompe la propiedad *safe*, se retorna un `String` detallando cuáles y en qué línea del código original se encuentran las operaciones que colisionan.

# 4. DETECCIÓN DE CANALES ABIERTOS

## 4.1. Motivación

Consideremos el programa en Go de la figura 4.1, que imprime por pantalla los primeros diez elementos de la sucesión de Fibonacci.

```
func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        c <- x
        x, y = y, x+y
    }
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}
```

*Fig. 4.1:* Ejemplo de programa en Go

En el método `main` se crea un canal asincrónico `c` de capacidad 10 y se lanza una go-rutina que ejecuta el método `fibonacci`. Éste envía sobre `c` los primeros 10 elementos de la sucesión. Luego en el método principal se itera sobre `c` a través de la instrucción `range`, que realiza una recepción de los elementos almacenados en su *buffer* y los imprime por pantalla.

La ejecución de este programa producirá un *deadlock*, debido a que cuando el canal `c` ya no tenga elementos en el *buffer*, el `receive` que ejecuta la instrucción `range` quedará sin sincronizar. Es necesario que la go-rutina `fibonacci` cierre el canal antes de que se ejecute un `range-over-channel`. Si bien en este caso la herramienta Gong detecta que no se cumple *liveness*, veremos en la sección 5.1 que el *deadlock* detectado no necesariamente se corresponde con el `recv` que se realiza dentro del `range`, y esto se debe a la incorrecta representación en MiGo que se eligió para definir el ciclo.

Muchos errores en Go pueden darse por no seguir correctamente los principios de co-

municación. Uno de ellos es el **Principio de cierre de canales**, que establece que la go-rutina que envía sobre un canal, debe cerrarlo para indicar que no se enviarán más valores sobre el mismo. No cumplir este principio puede producir *panic-failures* (enviar sobre un canal que el receptor cerró) o *deadlocks* (recibir sobre un canal que el emisor no cerró).

Por este motivo se decidió implementar un detector estático de canales abiertos en Gong. El objetivo consiste en identificar qué canales no fueron cerrados y en qué línea del código original se encuentran, de manera que el programador pueda validar que todas las go-rutinas emisoras hayan cerrado correctamente sus canales, manteniendo el principio de cierre de canales, y así evitar posibles *deadlocks*. Éste chequeo será estático (previo a la compilación) y se hará el análisis sobre la ejecución simbólica del tipo comportamental que representa al programa en Go.

## 4.2. Enfoque naive

Para identificar cuáles son los canales que no fueron cerrados y no se vuelven a utilizar, se podrían recorrer los estados finales del LTS generado buscando *buffers* abiertos. Que un estado sea final significa que no hay acciones de sincronización que le permitan transicionar a otro estado, por lo que podemos asegurar (por cómo se construye el LTS) que en el resto del programa ya no habrá instrucciones de comunicación sobre los canales afectados. Luego, para buscar los canales abiertos bastaría con recorrer el tipo que representa ese estado e identificar si hay alguna instrucción `Buffer` que tenga como primer argumento un *booleano* seteado en `true`, lo que significa que el canal asociado a ese *buffer* está abierto. Notar que no existen reglas de reducción que eliminen al constructor `Buffer`, por lo que una vez que un canal se crea, su *buffer* asociado debería estar presente en todos los estados subsiguientes.

Sin embargo, a la hora de analizar los LTSs generados para ciertos programas, los estados finales no fueron los esperados. Consideremos el ejemplo[1] de la figura 4.2.

Este programa crea dos canales `a` y `b`, envía un valor por `b` que es recibido en paralelo por la go-rutina `aux` y finalmente cierra `a` y recibe su valor por defecto. Dado que `b` nunca fue cerrado, se esperaría que en el estado final aparezca el tipo `Buffer b (True,0,0)`, ya que no hubo una acción de sicronización con este que le permita transicionar a otro estado. El GoType asociado al programa que generó Gong se puede ver en la figura 4.3.

A partir de este tipo se genera el LTS que representa la ejecución simbólica, y el re-

[1] Se obviaron los números de línea y las historias de ejecución propuestos en [6] con el objetivo de facilitar la legibilidad

```
def main() :
    let a = newchan ty, 0;
    let b = newchan ty2, 0;
    spawn aux(b);
    send b;
    close a;
    recv a;
def aux(ch) :
    recv ch;
```

*Fig. 4.2:* Ejemplo de programa en MiGo

```
EqnSys (bind [[(aux,{ChanAbst (bind [ch] (Recv 0@0 Null))})]]
(New 0 (bind a (New 0 (bind b (Par [ChanInst (TVar 2@0) [0@0],
Send 0@0 (Close 1@0 (Recv 1@0 Null))])))))))
```

*Fig. 4.3:* `GoType` generado para el programa de la figura 4.2

sultado podemos verlo en la figura 4.4.

En el estado 0 se definen los dos canales `a` y `b` y en paralelo se ejecutan los siguientes procesos: recibir sobre `b` (correspondiente a la go-rutina `aux`); enviar sobre `b`, cerrar `a` y recibir sobre `a` (correspondiente al proceso principal `main`); los *buffers* abiertos asociados a `b` y `a` respectivamente. Luego se realiza una sincronización sobre `b` (`Recv-Send`) y se transiciona al estado 1, en donde se sincronizan el `Close` sobre `a` con su `Buffer` asociado y genera una transición $\tau$ al estado 2. En este se sincroniza el `Recv` sobre `a` con su `Buffer` cerrado y transiciona a través de $a^\star$ hacia el estado final en donde ya no hay más tipos que puedan sincronizar.

Se puede observar que en el estado 3, a pesar de que es final, no aparece el *buffer* asociado al canal `b`. Esto sucede porque en la transición entre el estado 0 y el 1 desaparecen el constructor `New` que liga a `b` dentro del tipo `Par`, y el constructor del *buffer* asociado a `b`. Luego el canal `b` queda abierto, pero el constructor `Buffer b (True,0,0)` no pertenece al estado final.

Si bien no existen reglas de reducción que permitan eliminar estos constructores, para entender por qué sucede esto hay que observar las reglas de congruencia estructural de la figura 3.7. La regla $(\nu a)\lfloor a \rfloor \equiv \mathbf{0}$ establece que la creación de un canal que se liga en el tipo que contiene a su *buffer* asociado abierto, es congruente con el tipo vacío. En consecuencia, no se puede asegurar que un canal que no es cerrado y que no se vuelve a utilizar en el futuro de la ejecución, aparezca en el estado final. Podríamos intentar eliminar esa regla de congruencia de manera tal de poder implementar el análisis hecho previamente.

```
-- Estado 0:
EqnSys (bind [[(aux,{ChanAbst (bind [ch] (Recv 0@0 Null))})]]
(New (-1) (bind a6
    (New (-1) (bind b7
        (Par
            [Recv 0@0 Null,
            Send 0@0 (Close 1@0 (Recv 1@0 Null)),
            Buffer 0@0 (True,0,0),
            Buffer 1@0 (True,0,0)])))))))

-- Estado 1:
EqnSys (bind [[(aux,{ChanAbst (bind [ch] (Recv 0@0 Null))})]]
(New (-1) (bind a6
    (Par
        [Close 0@0 (Recv 0@0 Null),
        Buffer 0@0 (True,0,0)])))))

-- Estado 2:
EqnSys (bind [[(aux,{ChanAbst (bind [ch] (Recv 0@0 Null))})]]
(New (-1) (bind a3
    (Par
        [Recv 0@0 Null,
        Buffer 0@0 (False,0,0)])))))

-- Estado 3:
EqnSys (bind [[(aux,{ChanAbst (bind [ch] (Recv 0@0 Null))})]]
(New (-1) (bind a3
    (Par
        [Buffer 0@0 (False,0,0)])))))
```

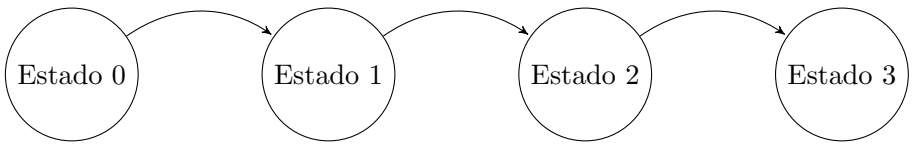


*Fig. 4.4:* LTS generado para el `GoType` de la figura 4.3

En Gong, la aplicación de la regla de congruencia a eliminar ocurre en la función `normalise`. Como se mencionó anteriormente, esta función es la encargada de "normalizar" un estado, generando una composición en paralelo de todos los tipos que están listos para sincronizar dentro de este e inicializando los *buffers*. Cada *buffer* es inicializado por la función `initiate`, que cada vez que se encuentra con un `New`, crea un constructor `Buffer` abierto asociado a ese canal. Luego, la función `gcBuffer` recorre todos los tipos y elimina los *buffers* repetidos y los de los canales que no se van a usar en el futuro, independientemente de si están cerrados o no. Finalmente, `normalise` invoca a la función `nf` que, entre otras cosas, elimina los constructores `New` de los canales que no se usan en el futuro. Entonces, al haber eliminado los tipos `Buffer`, también se eliminan los `New` correspondientes a esos canales.

El primer enfoque consistiría entonces en modificar la función `gcBuffer`, cambiando sus condiciones al momento de eliminar un *buffer*, de manera que el único motivo para esto sea que está cerrado y el canal no se utiliza en el futuro. Esto significa que el último estado del LTS de los programas que satisfacen *safety* y *liveness* debería ser `Null`, a menos que haya quedado algún canal abierto. Con esta idea, el LTS generado para el ejemplo de la figura 4.2 tendría en su estado final, únicamente al *buffer* de `b` abierto (el de `a` se elimina luego de la última sincronización sobre el mismo porque está cerrado y no se va a volver a utilizar).

Sin embargo, este enfoque es muy limitado en cuanto al conjunto de programas para los cuales se podrían detectar correctamente qué canales no fueron cerrados. Pensemos en programas de ejecución infinita, ¿qué sucede si el LTS no tiene estados finales?

Consideremos el código en MiGo de la figura 4.5. Este programa crea dos canales e itera infinitamente enviando y recibiendo sobre el segundo. El LTS generado por Gong puede verse en la figura 4.6.

El LTS cuenta con un único estado que transiciona a sí mismo a través de la sincronización de las acciones `Send` y `Recv` sobre `b` que se ejecutan en paralelo. En este caso el enfoque naive no detectaría que el canal `a` nunca fue cerrado ya que el autómata no posee estados finales para analizar.

```
def main() :
    let a = newchan ty, 0;
    let b = newchan ty2, 0;
    spawn s(b);
    spawn r(b);
def s(x) :
    send x;
    call s(x);
def r(x) :
    recv x;
    call r(x);
```

*Fig. 4.5:* Ejemplo de programa en MiGo

```
-- Estado 0:
EqnSys (bind
    [[(s1,{ChanAbst (bind [x]
        (Send 0@0 (Seq [ChanInst (TVar 1@0) [0@0],Null]))})),
    (r,{ChanAbst (bind [x]
        (Recv 0@0 (Seq [ChanInst (TVar  1@1) [0@0],Null]))})]]
(New (-1) (bind a7
    (New (-1) (bind b8
        (Par
            [Send 0@0 (ChanInst (TVar 2@0) [0@0]),
            Recv 0@0 (ChanInst (TVar 2@1) [0@0]),
            Buffer 0@0 (True,0,0),
            Buffer 1@0 (True,0,0)])))))))
```

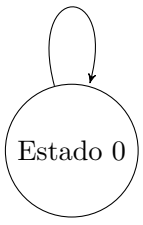


*Fig. 4.6:* LTS generado para el ejemplo de la figura 4.7 con la herramienta original

### 4.2.1. Consecuencias de modificar la congruencia estructural

Consideremos a continuación el programa de la figura 4.7. Tiene un método principal que crea dos canales `a` y `b` y lanza dos go-rutinas. La primera recibe sobre el canal `a`, crea un nuevo canal `c` y se llama infinitamente con `a` y `c` como parámetros. La segunda envía sobre `a` y se llama infinitamente. Notar que se crean infinitos canales (`c`) que nunca son cerrados.

```
def main():
    let a = newchan ty, 0;
    let b = newchan ty2, 0;
    spawn w(a);
    spawn r(a,b);

def r(x,y):
    let c = newchan ty3, 0;
    recv x;
    call r(x,c);

def w(x):
    send x;
    call w(x);
```

*Fig. 4.7:* Ejemplo de programa en MiGo

La herramienta sin las modificaciones propuestas, produce el LTS[2] de la figura 4.8. Consta de un estado inicial en donde se inicializan los 3 canales utilizados, y luego a través de la sincronización sobre `a` (`Send-Recv`) se avanza al estado 1. Este es similar al anterior, pero elimina la inicialización del canal `b` y transiciona a sí mismo sincronizando sobre `a`. Puede observarse que la ejecución simbólica es infinita, y por lo tanto, no hay estados finales.

Para este caso, con el enfoque naive se generaría sin embargo un LTS infinito y la ejecución de Gong no terminaría. Para entender por qué sucede esto hay que analizar cómo se normaliza el estado en donde se realiza la invocación a la función `r(x,y)`, en particular cómo se inicializan los *buffers*. La función `initiate` crea un constructor `Buffer` abierto asociado al canal `c` que se crea al inicio del método, y `gcBuffer` decide no eliminarlo ya que cumple con la condición de estar abierto. Luego se realiza la llamada recursiva, y se repite el mismo proceso para calcular el estado siguiente. Esto provoca que en cada invocación se genere un nuevo *buffer* asociado al canal `c` que se crea, y `unfoldTillGuard` no puede detener la ejecución porque los estados son distintos y no detecta que ya fueron recorridos, por lo que se siguen creando estados infinitamente.

[2] Sólo se muestra el `GoType` del estado para facilitar legibilidad

```
-- Estado 0:
New (-1) (bind a10
    (New (-1) (bind b11
        (New (-1) (bind c12
            (Par [Send 2@0 (ChanInst (TVar 3@1) [2@0]),
                 Recv 2@0 (ChanInst (TVar 3@0) [2@0,0@0]),
                 Buffer 0@0 (True,0,0),
                 Buffer 1@0 (True,0,0),
                 Buffer 2@0 (True,0,0)])))))))

-- Estado 1:
New (-1) (bind a14
    (New (-1) (bind c15
        (Par [Send 1@0 (ChanInst (TVar 2@1) [1@0]),
             Recv 1@0 (ChanInst (TVar 2@0) [1@0,0@0]),
             Buffer 0@0 (True,0,0),
             Buffer 1@0 (True,0,0)])))))
```

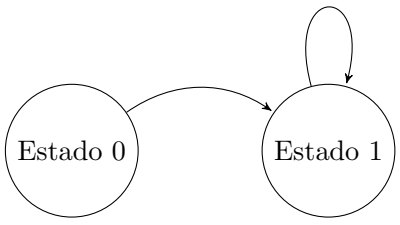


*Fig. 4.8:* LTS generado para el ejemplo de la figura 4.7 con la herramienta original

Puede observarse en este ejemplo el rol fundamental que tiene la congruencia estructural para poder generar LTS finitos de programas infinitos. En particular la regla $(\nu a)\lfloor a \rfloor \equiv \mathbf{0}$ es la que en este caso permite generar un estado que transiciona a sí mismo y representar una ejecución simbólica infinita.

## 4.3. Algoritmo de detección de canales abiertos

Se decidió entonces modificar la estrategia con el objetivo de solucionar los problemas del enfoque naive y evitar introducir no terminación en la generación de LTS para programas *fenced*. Para ello se mantiene la definición original de congruencia estructural de la figura 3.7.

Analicemos el algoritmo de generación de LTS que implementa Gong descripto en la sección 3.6.2.3. Primero se normaliza el `GoType` original (se aplican las reglas de congruencia estructural) y se lo agrega a una lista de estados a explorar. Para cada elemento de esa lista, Gong realiza los siguientes pasos:

1- Calcula los sucesores usando la función `genSuccs`

2- Normaliza cada sucesor

3- Agrega los no explorados a la lista de estados a explorar

El problema que surge al querer detectar canales abiertos es que en el paso 2, cuando se aplican las reglas de congruencia estructural, se eliminan las referencias a los canales que no fueron cerrados y que no se van a volver a utilizar.

La nueva estrategia consiste en generar el LTS como lo hace Gong originalmente y luego analizar los canales que se eliminan en el paso 2. Para esto, una vez construido el LTS, se recorren todos los estados, y para cada uno se computan los sucesores sin normalizar y se les aplica una función que detecta canales abiertos inutilizables.

La función que elimina los canales abiertos en el paso 2 es `gcBuffer`. Ésta invoca a `gcBufferList`, que toma una lista de `GoTypes`, que representa los tipos que se ejecutan en paralelo en un estado, y devuelve otra a la que se le eliminan los *buffers* asociados a los canales que no se vuelven a utilizar. Se procedió entonces a modificar dicha función para que devuelva además una lista de nombres de canales (`ChName`) que contiene los nombres de los canales asociados a *buffers* abiertos que fueron eliminados por la función `normalise`.

Con esta modificación, podemos entonces generar una nueva función `normalise'` que, en vez de devolver el estado normalizado, devuelve una lista de nombres de canales que no fueron cerrados y que fueron eliminados en el proceso de normalizarlo.

## 4.4. Pérdida de canales abiertos

Utilizando las reglas de semántica operacional de los tipos comportamentales junto con las de congruencia estructural (figuras 3.6 y 3.7 respectivamente) se puede definir la relación de sucesión de estados. Un estado $T'$ es sucesor de un estado $T$ ($T' \in sucs(T)$), si existe un estado intermedio $T''$ tal que $T \xrightarrow{\tau} T'' \equiv T'$.

Consideremos la relación de equivalencia estructural $\equiv'$ como la relación $\equiv$ definida en la figura 3.7 pero sin la regla $(\nu a)\lfloor a \rfloor \equiv \mathbf{0}$. Diremos que un estado $T$ *tiene un canal abierto sin usar* si se cumple que $\exists T'$ tal que $T \equiv' T' \mid (\nu a)\lfloor a \rfloor$ y $a \notin fn(T')$.

Luego definimos una nueva relación de sucesión de estados utilizando la nueva relación de equivalencia. Un estado $T'$ es sucesor de un estado $T$ ($T' \in sucs'(T)$), si existe un estado intermedio $T''$ tal que $T \xrightarrow{\tau} T'' \equiv' T'$.

Finalmente podemos definir la propiedad de pérdida de canales. Un estado T *pierde canales abiertos* si:

- $T$ *tiene un canal abierto sin usar*, o
- $\exists T' \in sucs'(T)$ tal que $T'$ *pierde canales abiertos.*

Como se mencionó en la sección 4.2.1, usar la relación de equivalencia $\equiv'$ puede provocar la no terminación en la construcción del LTS. Por este motivo, la propiedad de pérdida de canales se verifica para los estados del LTS que originalmente genera Gong utilizando la congruencia $\equiv$.

## 4.5. Implementación

En el archivo `OpenChannels.hs`[14] se encuentra la implementación del detector de canales abiertos. El algoritmo consiste en: recorrer todos los estados del LTS y para cada uno calcular los sucesores sin normalizar; si el estado es final (no tiene sucesores) se busca algún *buffer* abierto; caso contrario, se guardan los nombres de los *buffers* eliminados al normalizar los sucesores, y se prosigue con la recursión.

El método principal es `openChannels`. Éste toma como parámetro un entero que representa la cota, una lista de `Eqn` que representa los estados del LTS, y una lista de tuplas

de *Strings* que funcionará como acumulador para ir almacenando los canales abiertos (se inicializa vacía). El primer elemento de la tupla corresponderá al nombre del canal en el programa en MiGo, y el segundo a la línea en donde fue creado en el código original en Go. La función devuelve un *String* que detalla si hay canales abiertos o no, y en qué línea se crearon.

`openChannels` itera sobre la lista de estados del LTS. El primer paso consiste en quedarse con el `GoType` del estado que representa el momento de la ejecución (es decir, descartar las definiciones de las funciones auxiliares). Luego se procede a calcular sus sucesores con la función `genSuccs`.

Si la lista de estados sucesores es vacía, se obtienen los nombres de los canales cuyos *buffers* están abiertos en ese estado, y se llama recursivamente agregando los nuevos elementos al acumulador. La función que obtiene los canales abiertos se llama `getOpenChannelsFinal State`, y tiene como argumentos un `GoType` y una lista en donde se acumulan los canales. Por cómo se construyen y se normalizan los tipos dentro de los estados, sabemos que todas las inicializaciones de canales (constructures `New`) se encuentran al principio. Esta función recorre los `New` dentro del estado, y para cada uno de ellos busca su constructor `Buffer` correspondiente. Si el *buffer* está abierto, agrega al acumulador el nombre del canal con su número de línea, y se llama recursivamente con el tipo que prosigue luego del `New`; caso contrario el acumulador se mantiene sin modificaciones. Una vez que encuentra un tipo que no tiene la forma `New`, devuelve el acumulador. Para saber si existe un *buffer* abierto para un canal determinado, se llama a la función `anOpenBuffer`. Ésta recibe el tipo y el nombre del canal, y recorre recursivamente el tipo hasta encontrar una instrucción `Buffer`. Si el nombre asociado al *buffer* coincide con el pasado como parámetro, se devuelve el *booleano* que representa si está abierto o cerrado, y si no se devuelve *false*.

Si la lista de sucesores no es vacía, se le aplica a cada elemento de la lista la función `normalise'` definida en la sección 4.3. Ésta devuelve los nombres de los canales cuyos *buffers* fueron eliminados y estaban abiertos. Para cada uno se crea una tupla con el número de línea donde se crearon, se agrega al acumulador, y se llama recursivamente para recorrer el resto de los estados del LTS.

Para encontrar en qué línea se inicializa un canal, hay que tener en cuenta no sólo la ejecución principal sino también las definiciones de las funciones auxiliares, ya que al normalizar los tipos los números de línea en los constructores `New` de los canales que no se crean en el método principal son borrados. Para esto entonces se definió la función `getGoTypesList` que devuelve una lista de `GoTypes` en donde se encuentran los tipos de las funciones auxiliares y el de la ejecución principal. Luego, la función `getOpenChannelsLine`

recorre esa lista de tipos junto con la lista de canales eliminados, buscando para cada uno su constructor `New` para obtener el número de línea.

Finalmente, la función `openChannels`, una vez que terminó de recorrer la lista de estados, devuelve el número de línea en donde se crearon los canales que fue almacenando en el acumulador. El motivo por el cual no se devuelve el nombre de los canales abiertos, es que éstos se pierden al realizar la conversión del código original a MiGo. Los nombres de los canales en el GoType no se corresponden con los del código en Go, y esto se debe al renombre y a la reasignación de las variables que se realiza en el extractor de SSA. Se puede observar en la figura 4.9 el resultado de correr Gong con estas modificaciones sobre el programa de Go que calcula el factorial de la figura 2.1.

```
Liveness: Term is live

Safety: Term is Safe

Open Channels: Open channels created at lines 2 and 11
```

*Fig. 4.9:* Output de Gong con el código de la figura 2.1

# 5. LIMITACIONES DE DINGO HUNTER Y GONG

## 5.1. Tipo comportamental para `range-over-channel`

Consideremos el código de la figura 5.1. Este programa crea dos canales `a` y `b` y ejecuta en paralelo las acciones de: enviar dos veces sobre `a`, cerrar `a` y enviar sobre `b`, y recibir sobre `a` (usando `range`) y recibir sobre `b`. Notar que la comunicación entre canales es correcta y que se satisfacen las propiedades de *liveness* y *safety*.

```
1   func main() {
2       a := make(chan int, 0)
3       b  := make(chan int, 0)
4       go rec(a, b)
5       a <- 1
6       a <- 2
7       close(a)
8       b <- 1
9   }

10  func rec(a chan int, b chan int){
11      for elem := range a {
12          fmt.Println(elem)
13      }
14      fmt.Println(<-b)
15  }
```

*Fig. 5.1:* Ejemplo de programa en Go

En la figura 5.2 podemos ver el código en MiGo después de correr la herramienta Dingo Hunter. La instrucción `range-over-channel` se representa en MiGo utilizando la elección interna `if-then-else`, en donde la primera rama es una llamada recursiva para volver a ejecutar el `recv`, y la otra es la ejecución posterior al `range`. Por lo que la función `rec` del código original se traduce en: recibir sobre `a` y luego, llamarse recursivamente o recibir sobre `b`.

En las figuras 5.3 y 5.4 podemos ver una versión simplificada del LTS generado por Gong. Cada estado está representado como una lista de tipos que se ejecutan en paralelo. En cada tipo se detalla el número de línea para distinguir con qué instrucción del código original se mapea.

```
def main():
    let t0 = newchan main.t0_0_0, 0 @2;
    let t1 = newchan main.t1_0_0, 0 @3;
    spawn rec(t0, t1) @4;
    send t0 @5;
    send t0 @6;
    close t0 @7;
    send t1 @8;
def rec(a, b):
    recv a @12;
    if call rec(a, b) @0;
    else recv b @14;
    endif;
```

*Fig. 5.2:* Código MiGo para el ejemplo de la figura 5.1

En el estado inicial se encuentran en paralelo los tipos correspondientes al método principal, a la función `rec` y la inicialización de los *buffers* asociados a ambos canales. Como mencionamos anteriormente, el `range` sobre `a` se representa como un `Recv` seguido de una elección interna `IChoice` cuyas ramas son el llamado recursivo y lo que sigue después (recibir sobre `b`). La transición al estado 1 se realiza mediante la sincronización sobre el canal `a`. Este estado tiene como sucesores a aquellos que se corresponden con ejecutar las dos ramas del `IChoice`. Se trancisiona al estado 2 a través de la primera rama invocando a la función `rec(a,b)` y reemplazando su definición, y al estado 3 a través de la segunda rama (`Recv b`).

En el estado 3 se produce un *deadlock*. No hay acciones posibles de sincronización ya que los tipos preparados para sincronizar son recibir sobre `b` y enviar sobre `a`, y como el canal `b` está abierto tampoco puede sincronizar con el `Recv` enviando el valor por defecto.

Desde el estado 2 se transiciona hacia el estado 4 sincronizando sobre el canal `a`, con la instrucción `send` de la línea 6 y el `recv` de la línea 12. El estado 4 presenta de nuevo dos transiciones correspondientes a ejecutar ambas ramas del `IChoice`. La primera es hacia el estado 5 invocando a la función `rec(a,b)`, y la segunda hacia el estado 6 ejecutando el `Recv b`. Éste último transiciona hacia el estado 9 sincronizando el `Close` sobre el canal `a` con su *buffer* asociado que estaba abierto. Éste pasa a cerrarse y a eliminarse de la lista ya que no se vuelve a usar en esa rama de la ejecución.

El estado 5 transiciona hacia el 7 realizando una sincronización sobre el `Recv a` y el `Close a`, y actualiza el estado del *buffer* asociado a cerrado. El estado 7 de nuevo presenta dos opciones de transición ejecutando ambas ramas del `IChoice`. La primera es la invo-

```
-- Estado 0:
[Recv "12" a (IChoice (ChanInst rec(a,b)) (Recv "14" b)),
Send "5" a (Send "6" a (Close "7" a (Send "8" b ))),
Buffer b (True,0,0), Buffer a (True,0,0)]

-- Estado 1:
[IChoice (ChanInst rec(a,b)) (Recv "14" b),
Send "6" a (Close "7" a (Send "8" b)),
Buffer b (True,0,0), Buffer a (True,0,0)]

-- Estado 2:
[Recv "12" a (IChoice (ChanInst rec(a,b)) (Recv "14" b)),
Send "6" a (Close "7" a (Send "8" b)),
Buffer b (True,0,0), Buffer a (True,0,0)]

-- Estado 3 (deadlock):
[Recv "14" b,
Send "6" a (Close "7" a (Send "8" a)),
Buffer b (True,0,0), Buffer a (True,0,0)]

-- Estado 4:
[IChoice (ChanInst rec(a,b)) (Recv "14" b),
Close "7" a (Send "8" b),
Buffer b (True,0,0), Buffer a (True,0,0)]

-- Estado 5:
[Recv "12" a (IChoice (ChanInst rec(a,b)) (Recv "14" b)),
Close "7" a (Send "8" b),
Buffer b (True,0,0), Buffer a (True,0,0)]

-- Estado 6:
[Recv "14" b,
Close "7" a (Send "8" b Null),
Buffer b (True,0,0), Buffer a (True,0,0)]

-- Estado 7:
[Send "8" b,
IChoice (ChanInst rec(a,b)) (Recv "14" b),
Buffer b (True,0,0), Buffer a (False,0,0)]

-- Estado 8:
[Send "8" b,
Recv "12" a (IChoice (ChanInst rec(a,b)) (Recv "14" b)),
Buffer b (True,0,0), Buffer a (False,0,0)]

-- Estado 9:
[Send "8" b,
Recv "14" b,
Buffer b (True,0,0)]

-- Estado 10:
[Buffer b (True,0,0)]
```

*Fig. 5.3:* Estados del LTS generado para el MiGo de la figura 5.2

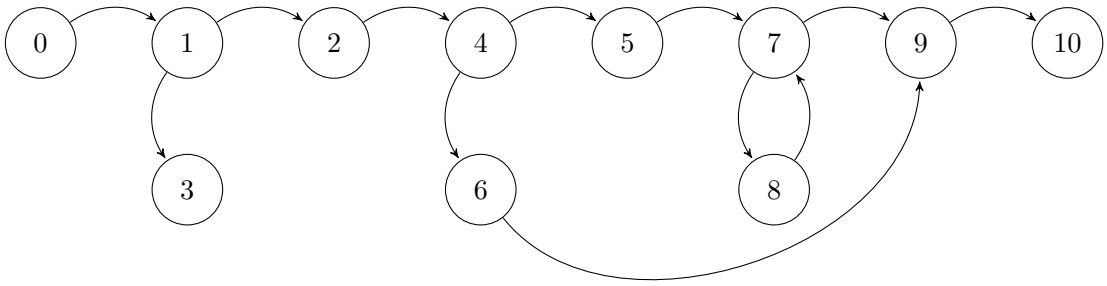


*Fig. 5.4:* LTS generado para el MiGo de la figura 5.2

cación a `rec(a,b)` mediante la cual transiciona al estado 8. En este estado se sincroniza el `Recv a` con el *buffer* asociado a `a` que está cerrado, y se vuelve al estado 7. La segunda opción del `IChoice` del estado 7 es transicionar hacia 9. En este se sincronizan el `send` sobre `b` de la línea 8 con el `recv` de la línea 14, y se avanza hacia el estado final 10 que solo posee el *buffer* asociado a `b`.

En este programa se nota claramente por qué es importante cerrar el canal `a`. Si no existiera la instrucción `Close a`, el estado 5 no podría transicionar porque no existiría una acción que sincronice con el `Recv a` de la línea 12 (el correspondiente al `range`) y se produciría un *deadlock*.

Correr la herramienta Gong sobre este ejemplo devolverá que el programa no cumple con la propiedad de *liveness* ya que detecta un *deadlock* en el estado 3. Y si bien es correcto que el *deadlock* se produce en el código MiGo, no es verdad que ocurre en el código original de Go. Esto sucede porque el `if-then-else` de MiGo no representa la ejecución real del `range-over-channel`. Como Gong asume que ambas ramas del `IChoice` pueden ejecutarse en cualquier momento, se permite salir del `range` a pesar de que éste no haya terminado de leer todos los valores que se enviaron sobre el canal, produciendo así un *deadlock* que no se produciría en el código original.

Se podría pensar en modificar Dingo Hunter de manera de representar al `range` utilizando una elección externa (`select`). Para este ejemplo podría pensarse en el Gotype `OChoice [Recv a (ChanInst rec(a,b)), Recv b]`, en donde se va a ejecutar el `Recv a` siempre que se pueda (sincronizando con los primeros envíos sobre `a`) y luego se ejecuta la segunda rama sincronizando con el envío sobre `b` y termina. Correr Gong sobre esta modificación devuelve correctamente que el programa es *live*. Sin embargo, si sacáramos

la instrucción `close` de la línea 7, la sincronización con el `select` seguiría funcionando y el programa seguiría siendo *live*, lo que no se corresponde con la ejecución real ya que el `range-over-channel` sobre un canal abierto produce *deadlock*.

Además esta codificación tampoco funcionaría siempre. Supongamos que tenemos los siguientes GoTypes ejecutándose en paralelo:

```
[Send a (Send a (Send a (Close a))),
Send b (Close b),
OChoice [Recv a (ChanInst rec(a,b)), Recv b)]]
```

En este caso cualquiera de las dos ramas del `OChoice` podría ejecutarse porque ambas están listas para sincronizar, pero una vez que se realice la sincronización sobre `b` reduciendo la segunda rama, quedarán envíos sobre `a` sin sincronizar y el programa no será *live*, a pesar de que el código original lo es.

## 5.2. Representación de MiGo para ciclos

Como se mencionó en la sección 2.3, SSA representa los ciclos a través de etiquetas y de jumps. Esto provoca una limitación al querer analizar ciclos acotados. Consideremos el ejemplo de la figura 5.5. Este programa crea un canal asincrónico, envía 3 valores, y luego los recibe en un ciclo que itera 3 veces.

```
1   func main() {
2       a := make(chan int, 3)
3       a <- 1
4       a <- 2
5       a <- 3
6       for n := 0; n <= 3; n++ {
7           fmt.Println(<-a)
8       }
9   }
```

*Fig. 5.5:* Ejemplo de un ciclo en Go

Podemos ver el código traducido a MiGo en la figura 5.6. Dingo Hunter utiliza la representación intermedia SSA para extraer el código de los ciclos, por lo que utiliza invocaciones recursivas. El ciclo está definido por la función `main3`, allí se realizan las acciones de recibir y llamarse recursivamente 3 veces (se le agrega un parámetro extra para representar que no es la invocación a la función original). En este punto ya podemos identificar como un problema que se quiera trabajar con ciclos con cotas muy grandes (implicaría un

código muy extenso).

```
def main():
    let t0 = newchan main.t0_0_0, 3 @2;
    send t0 @3;
    send t0 @4;
    send t0 @5;
    call main#3(t0) @0;
def main#3(t0):
    recv t0 @7;
    call main#3(t0, t0) @0;
    recv t0 @7;
    call main#3(t0, t0) @0;
    recv t0 @7;
    call main#3(t0, t0) @0;
```

*Fig. 5.6:* Codigo MiGo para el ejemplo de la figura 5.5

Este programa no puede ser analizado por Gong porque genera un `GoType` en donde aparece el tipo `Seq` con más de dos elementos en la lista y el segundo no es `Null`. Podemos observarlo en la figura 5.7. La salida al correr Gong sobre este programa es un error con el mensaje "We don't deal with full Seq yet".

```
EqnSys (bind [[(main#3,{ChanAbst (bind [t0] (Recv "7" 0@0 (Seq "" [ChanInst
(TVar "" 1@0) [0@0,0@0],Recv "7" 0@0 (Seq "" [ChanInst (TVar "" 1@0) [0@0,
0@0],Recv "7" 0@0 (Seq "" [ChanInst (TVar "" 1@0) [0@0,0@0],Null])])])))})]]
(New "2" 3 (bind t0 (Send "3" 0@0 (Send "4" 0@0 (Send "5" 0@0 (Seq ""
[ChanInst (TVar "" 1@0) [0@0],Null]))))))))
```

*Fig. 5.7:* GoType para el MiGo de la figura 5.6

Otro problema que surge al representar los ciclos es cuando dentro de éste se crea un canal. En esos casos, el llamado recursivo agrega un parámetro extra con el canal en cada invocación, y luego queda invalidado por tener una cantidad excesiva de parámetros.

Por último, los ciclos acotados anidados tampoco se representan de manera de lograr un comportamiento real del programa. En estos casos se realizan los llamados recursivos del ciclo interno y se ignoran los del externo.

En ejemplos como el de la figura 5.5 podría ser de utilidad saber cuándo un canal está cerrado para garantizar que un ciclo termine. Si el ciclo iterara una cantidad de veces mayor a la cantidad en envíos que se realizan sobre `a`, se produciría un *deadlock* porque

habría acciones de recepción sin sincronizar. Sin embargo, si el canal estuviese cerrado, esto no sucedería porque se recibiría el valor por defecto del tipo del canal. La detección de canales abiertos podría utilizarse en conjunto con una regla de tipado más adecuada para ciclos acotados de manera de garantizar la ausencia de *deadlocks* para estos casos.

# 6. CONCLUSIONES

Este trabajo tuvo como objetivo detectar cuándo no se cumple uno de los principios de comunicación en Go, en particular el principio de cierre de canales, realizando un análisis estático para identificar canales abiertos. Para esto se extendieron las herramientas Dingo Hunter y Gong, que realizaban un chequeo estático de *liveness* y *safety* utilizando tipos comportamentales, agregando un análisis estático de detección de canales que no se cierran y que no se vuelven a utilizar. De esta manera se enriquecen las herramientas, ya que se agrega en la salida información sobre la línea del código en donde se inicializa un canal que nunca se cierra.

El chequeo se basó en el análisis del *labelled transition system* que denota la ejecución simbólica de los tipos comportamentales que representan acciones de comunicación del programa. Buscar los canales abiertos parecía inmediato: había que recorrer los estados del LTS e identificar *buffers* abiertos asociados a canales que no se volvieran a utilizar. Sin embargo este enfoque era muy limitado en cuanto al subconjunto de programas para los cuales funcionaba, por lo que tuvo que hacerse un análisis más exhaustivo sobre cómo se construía el LTS para lograr identificar correctamente los canales abiertos.

En este análisis se encontraron limitaciones de las herramientas a la hora de representar la ejecución real de los programas. Había casos que directamente no se contemplaban (ciclos acotados) o que se analizaban de manera errónea y generaban falsos positivos (`range-over-channel`). Si se tiene en cuenta que los ciclos son parte fundamental de la programación, este es un limitante muy grande para poder lograr un uso real de la aplicación. A esto se le agrega la restricción de que los programas deben ser *fenced*.

El trabajo a futuro para seguir enriqueciendo estas herramientas es muy amplio. Tanto el lenguaje MiGo como los Gotypes deberían extenderse para lograr una representación más precisa de la ejecución de los programas en Go, además de incorporar aquellas instrucciones que no están contempladas. También se podría pensar en una manera de optimizar los tiempos de ejecución de las herramientas. Si bien no fue un tema que se haya abarcado en este trabajo, existen casos en donde calcular el LTS puede llevar minutos, y esto se debe no solo a la cantidad de estados, sino a que Haskell es muy ineficiente para concatenar cadenas de texto, y agregar las historias de ejecución que se proponen en [6] puede ser muy costoso.

Otra posible extensión sería agregar en la salida un mensaje de error con información sobre la historia de ejecución al momento de detectar que un canal que no se vuelve a

utilizar no se cierra. Esto también requeriría mostrar ejecuciones infinitas, y es un problema que el trabajo [6] no resuelve.

## Bibliografía


[1] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. *Language primitives and type discipline for structured communication-based programming*. In European Symposium on Programming, pages 122-138. Springer, 1998

[2] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[3] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. *Fencing off go: Liveness and safety for channel-based programming*. In ACM SIGPLAN Notices, volume 52, pages 748-761. ACM, 2017.

[4] `https://github.com/nickng/dingo-hunter`

[5] `https://github.com/nickng/gong`

[6] Damián Furman. *Generación de mensajes de error significativos en herramienta de detección estática de deadlocks para programas en Go*. Tesis de Licienciatura.

[7] `https://go101.org/article/channel-closing.html`

[8] R. Milner. *Communication and Concurrency*. C.A.R. Hoare Series Editor. Prentice Hall, 1989.

[9] `https://blog.golang.org/share-memory-by-communicating`

[10] `https://golang.org/doc/effective_go.html`

[11] `https://godoc.org/golang.org/x/tools/go/ssa`

[12] Stephanie Weirich, Brent A. Yorgey and Tim Sheard *Binders Unbound*. September 2011.

[13] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[14] `https://github.com/dmarottoli/gong`