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

# Generación de mensajes de error significativos en herramienta de detección estática de *deadlocks* para programas en Go

Tesis de Licenciatura en Ciencias de la Computación

Damián Ariel Furman

Director: Hernán Melgratti

Buenos Aires, 2019

# GENERACIÓN DE MENSAJES DE ERROR SIGNIFICATIVOS EN HERRAMIENTA DE DETECCIÓN ESTÁTICA DE *DEADLOCKS* PARA PROGRAMAS EN GO

Go es un lenguaje de programación que incorpora la comunicación a través del intercambio de mensajes dentro de su *set* básico de instrucciones. Lange, Ng, Toninho y Yoshida proponen un sistema de tipos sesión para analizar propiedades sobre aspectos de comunicación en programas Go, como por ejemplo ausencia de deadlocks parciales y errores de comunicación. Esta técnica ha sido implementada en dos herramientas, **Dingo-Hunter** y **Gong**, que toman un programa Go e indican si cumple o no con estas propiedades. Sin embargo, su implementación no aporta más información que pueda servir al usuario para mejorar su código si existe algún problema. El objetivo del siguiente trabajo es mejorar la usabilidad de la herramienta incorporando *feedback* que pueda ser usado para resolver el problema encontrado. Para esto, se desarrolla una modificación de la implementación del tipo sesión que genera trazas de las posibles ejecuciones de los programas a analizar. Luego, cuando un programa no satisface alguna de las propiedades mencionadas, se indica qué instrucción genera el problema, en qué número de línea y cual es la traza de la ejecución que la generó, indicando invocaciones a funciones y sincronizaciones (envío y recepción correctos de un mensaje) previas.

**Palabras claves:** Go, Comunicación por intercambio de mensajes, Tipo Sesión, Liveness, Safety, Deadlock, Traza, Historia de ejecución.

# MEANINGFUL ERROR MESSAGES IN STATIC DEADLOCK DETECTION TOOL FOR GO PROGRAMS

Go is a programming language that incorporates comunication through message passing as part of its basic set of instructions. Lange, Ng, Toninho and Yoshida popose a session type's system for analyzing propertives over Go programs like absence of partial deadlocks or comunication errors. However, their implementation doesn't provide more information that could be used by the user of the tool to improve her code if there is any problem. The objective of the following work is to improve the usability of the tool incorporating feedback that may be used to solve problems found. With this purpose, a modification of the session types implementation is developed to generate traces of all possible executions of programs being analyzed. Then, when a program doesn't satisfy any of the properties mentioned above, the instruction that generates the problem is given, along with its line number and the trace of the execution that generated it, indicating previous function invocations and synchronization operations (sending and receiving messages).

**Keywords:** Go, Comunication through message passing, Session Types, Liveness, Safety, Deadlock, Trace.

# AGRADECIMIENTOS

A todes les que defienden la educación pública ya que estoy absolutamente convencido que sin ellos no tendría carrera de la cual licenciarme. En particular, a mis compañeres de militancia del Partido Obrero que me acompañaron en todas las luchas, marchas, ocupaciones piquetes y asambleas, en las buenas y en las malas sin dudar en ponerse siempre del lado del oprimido contra el opresor.

A mis amigos y amigas que me enseñaron lo que es la amistad y lo lindo que es el mundo cuando uno está en un abrazo de bola. A todos los que estuvieron cuando los necesitaba y también cuando no los necesitaba pero simplemente queríamos compartir una cerveza.

A mis compañeres de la facultad con quienes nos gastamos los ojos estudiando para parciales y finales y los dedos haciendo trabajos prácticos. Especialmente a todos los que entienden que el conocimiento es una construcción colectiva y vale cien veces más si se comparte. El esfuerzo hubiera sido cien veces mayor sin ustedes.

A mis viejos que insistieron desde siempre para que estudie en la universidad pública y que me alentaron y apoyaron incansablemente para que logre mis objetivos siempre.

A mi hermano que fue mi primer amigo y con quien compartí la mayor parte de mi vida.

A Hernán, mi director, que me enseño cómo hacer un trabajo de investigación, me orientó desde el instante que empecé ésta tesis, me ayudó siempre que lo necesité y sobre todo, me tuvo una paciencia infinita frente a las exigencias de plazos. Nada de esto hubiera sido posible sin él.

A mi amor, Lu, sin la cual la vida no tendría sentido. Ella hace que los fracasos se sientan con mil millones de veces menos intensidad y los éxitos con mil millones de veces más. De ella viene toda la fuerza y todo el cariño. Ella me enseño a luchar no sólo por mi sino por todos: por la educación, por la ciencia y por la humanidad.

A todes ustedes, gracias.

*A mi Compañera.*

# Índice general

# 1. INTRODUCCIÓN

A partir de los años 90 han surgido sistemas de tipado llamados tipos sesión[1] [3] que asignan un tipo que describe las comunicaciones que occurren en un programa; el sistema de tipos garantiza que el comportamiento de un programa bien tipado goza de determinadas propiedades, como por ejemplo, la ausencia de *deadlocks*. A partir del desarrollo masivo de los sistemas distribuidos y la programación concurrente, surgieron trabajos que proponen tipos sesión para garantizar propiedades en distintos lenguajes de programación.

En el año 2007, a su vez, surgió Go, un lenguaje de programación cuyo objetivo explícito es lograr una buena performance mediante el tipado estático, una alta legibilidad y una alta *performance* en operaciones sobre redes, multiprocesamiento y operaciones concurrentes [4]. Go se plantea dar todas las garantías posibles que ofrece un sistema de tipos fuerte dejando de lado la carga de complejidad que estos traen consigo al momento de programar.

A pesar de que en el último tiempo se han registrado avances en la detección de errores de comunicación, tales como deadlocks, race-conditions o uso no permitido de canales, aún no existe una técnica que permita identificarlos en cualquier caso y para cualquier tipo de código. El compilador de Go brinda algunas herramientas como un detector de *race-conditions* e incluso puede detectar los *deadlocks* en tiempo de ejecución si observa que el programa en términos globales no progresa (todos los hilos de ejecución están dormidos). Sin embargo, para poder detectarlo, el *deadlock* tiene que suceder primero, no puede ser prevenido. A su vez, sólo detecta un *deadlock* cuando *todos* los hilos de ejecución se detuvieron, por lo tanto nunca puede encontrar un *deadlock* parcial en sólo un subconjunto de las rutinas de ejecución. Si bien han surgido herramientas que permiten un testeo exhaustivo (por ejemplo, corriendo cientos de veces un código con distintos parámetros esperando encontrar si en alguna de esas ejecuciones se encuentra un problema), hasta el momento, no se conoce una técnica infalible que permita detectar cualquier tipo de problema de comunicación en cualquier caso.

Go es, a su vez, un lenguaje que crece año a año en popularidad y uso[2] debido a su buena performance combinada con un estilo de programación simple y legible. A su vez, incorpora la comunicación a través de canales como uno de sus tipos básicos para el cual no es necesario siquiera importar una librería. También incorpora la composición de un hilo de ejecución en paralelo como parte de su set básico de instrucciones con *threads* optimizados especialmente para ser muy ligeros. Esto lo convierte en un lenguaje ideal para la programación concurrente. Sin embargo, los problemas de concurrencia (y por lo tanto, de manera inversa, también la falta de herramientas para encontrarlos automáticamente) limitan el potencial del lenguaje. Existe incluso una polémica sobre priorizar las primitivas de sincronización por sobre la utilización de canales debido a la complejidad que los canales pueden agregar en ciertos casos si se los quiere utilizar de manera segura [6]. Por lo que la detección automática de errores y la verificación automática de programas pueden

[1] *session types*

[2] Según las estadísticas publicadas en el blog de Golang, este lenguaje incrementó su uso en proyectos de GitHub en un 52 % entre 2016 y 2017 [5]

impactar fuertemente en el propio paradigma del lenguaje y la forma en que sus usuarios programan.

Por ese motivo, los tipo sesión pueden ser un gran aporte al desarrollo de este lenguaje porque podrían permitir la detección de una gran cantidad de errores de programación en tiempo de compilación sin agregar ninguna complejidad al lenguaje[3]. Con esta perspectiva, Lange, Ng, Toninho y Yoshida, en un trabajo publicado en el 2017 [8], desarrollan una implementación de tipo sesión para la detección de *deadlocks* y de errores generados por no respetar la semántica de los canales de Go, que generan un error en tiempo de ejecución cuando se cierra un canal dos veces o se envía un mensaje sobre un canal cerrado. Las dos herramientas desarrolladas, **Dingo-Hunter** [9] y **Gong** [10], implementan la extracción de un lenguaje llamado MiGo que representa el comportamiento comunicacional del programa original (es decir, las estructuras de control y las operaciones de creación y cierre de canales, envío o recepción de mensajes) y la extracción del tipo y su consiguiente ejecución simbólica para verificar las propiedades, respectivamente.

En la figura 1.1 se puede observar un programa sencillo en Go que crea un canal sincrónico (tiene un buffer de capacidad 0) en la línea 4, trata de enviar primero sobre éste (línea 5) y luego recibir (línea 6), pero genera un *deadlock* en la línea 5 debido a que el envío del primer mensaje no sincroniza. En la figura 1.2 se ve el resultado de ejecutar las herramientas **Dingo-Hunter** y **Gong** sobre el programa de la figura 1.1. Estas devuelven un *booleano* por cada propiedad verificada indicando si se cumple con la propiedad o si esta no puede ser garantizada. Pero la herramienta no proporciona información sobre los estados del programa que son sospechosos, ni cómo estos pueden ser alcanzados.

Si bien **Dingo-Hunter** y **Gong** cumplen con un primer objetivo de detección de errores en tiempo de compilación, el resultado no otorga ninguna herramienta al programador para entender dónde está el error o cómo corregirlo. En este sentido, tienen una usabilidad con poco desarrollo que puede generar una mala experiencia en el usuario y llevarlos al desuso.

En el siguiente trabajo se propone extender la herramienta con mensajes de error significativos para el usuario. En este sentido, ante la presencia de un error en tiempo de ejecución o un posible *deadlock*, en lugar de simplemente indicar que puede existir un error, se provee al usuario con un ejemplo de ejecución que exhibe el problema detectado. En el trabajo, llamamos *traza* a tal ejemplo de ejecución. La traza identifica en primer lugar la instrucción que genera el fallo y muestra luego en orden inverso a la ejecución, las invocaciones a funciones realizadas por los distintos threads y las sincronizaciones sobre canales que ocurren. Ésta se construye en dos dimensiones: vertical para las operaciones realizadas en el mismo hilo de ejecución y horizontal para operaciones en hilos de ejecución separados (que se juntan en una misma historia a partir de una operación de sincronización). En la figura 1.3 se muestra el resultado de ejecutar nuevamente **Dingo-Hunter** y **Gong** sobre el programa de la figura 1.1 luego de las modificaciones realizadas.

[3] Esta opinión es compartida por Lange que reconoce el modelo de comunicación en Go como beneficioso principalmente porque permite desarrollar la verificación automática de ausencia de errores de comunicación [7]

```
1. package main
2.
3. func main() {
4.         ch := make(chan int)
5.         ch <- 1
6.         <- ch
7. }
```

*Fig. 1.1:* Ejemplo código en Go con deadlock en la línea 5

```
Liveness: False
Safety: True
```

*Fig. 1.2:* Resultado de ejecutar Dingo-Hunter y Gong sobre el programa de la figura 1.1

Para obtener la historia de las posibles ejecuciones fallidas es necesario primero acumular información sobre las historias de *todas* las posibles ejecuciones y luego mostrar la historia de una ejecución particular al detectar que esta ejecución falla. También, para el caso de la propiedad de comunicación segura (ausencia de errores en tiempo de ejecución), se identifica de qué manera se rompe esta propiedad (si es mediante el cierre de un canal más de una vez o si es debido al envío de un mensaje por un canal cerrado).

El objetivo del trabajo realizado es mejorar la usabilidad de las herramientas, por lo que también se detectaron y corrigieron algunos errores en la implementación en la medida que fueron encontrados. Finalmente, se proponen posibles trabajos futuros con el objetivo de continuar la mejora en la usabilidad.

```
There is a Send operation without synch on line 5
```

*Fig. 1.3:* Resultado de ejecutar Dingo-Hunter y Gong sobre el programa de la figura 1.1 luego de las modificaciones realizadas

# 2. ANTECEDENTES

En el siguiente capítulo se desarrollan los antecedentes considerados para el trabajo realizado. En la sección 2.1 se presentan aspectos fundamentales de la comunicación a través de canales en Go. En la sección 2.2 se repasa la historia de los tipos sesión. En la sección 2.3 se aborda el trabajo sobre el tipo sesión para MiGo, el lenguaje que abstrae el comportamiento comunicacional de Go, propuesto en [8]. Por último, en la sección 2.4 se describe la implementación de las herramientas **Dingo-Hunter** y **Gong**.

## 2.1. Modelo de comunicación en Go[1, 2]

Go se distingue, entre otras cuestiones, por incorporar canales como un tipo estándar dentro del lenguaje e incorpora la comunicación sincrónica y asincrónica a través de éstos mediante la operación (<-) que, según se escriba a derecha o a izquierda de la variable que representa al canal, implicará una operación de enviar o recibir un determinado valor. La operación `a <- ch`, por ejemplo, representa a la acción de recibir un valor por el canal ch y guardarlo en la variable a. La operación `ch <- 1`, por el contrario, representa a la acción de mandar 1 por el canal ch.

A la vez Go también ofrece *goroutines* incorporadas a las funcionalidades estandar, que pueden usarse sin incorporar ninguna librería. Las *goroutines* son *threads* livianos que no requieren interacción con el sistema operativo sino que son administradas por el *scheduler* de Go. Esto les permite ejecutar consumiendo un mínimo de recursos y explotar las ventajas de paralelizar la ejecución. Una *goroutina* se puede lanzar mediante el comando `go` seguido de una llamada a una función. La función invocada mediante el comando `go` se ejecutará en un nuevo hilo de ejecución lanzado en paralelo a la ejecución principal. El contexto de ejecución de esa función será el mismo contexto que tendría la función si se hubiera invocado normalmente desde la misma línea de ejecución (comparte variables globales con el thread principal, se le instancian los valores pasados por parámetro).

Distintas *goroutinas* pueden compartir un canal ya sea mediante el acceso común a la variable que lo referencia o mediante el pasaje de la referencia como parámetro de función o incluso a través de otro canal. Go no establece un límite a la cantidad de *goroutinas* que pueden utilizar el canal en simultáneo ni fuerza a especificar cuáles serán las *goroutinas* que utilicen el canal. Un canal puede pasarse a una cantidad indefinida de funciones o *goroutinas*.

La creación de un canal se realiza mediante la primitiva `make`, la cual se utiliza para crear distintas estructuras de datos como listas o mapas y toma como primer parámetro obligatorio el tipo a crear y como segundo parámetro opcional, la capacidad de la estructura. Si no se le pasa un segundo parámetro, la capacidad es cero. Los canales son siempre tipados: el tipo de un canal se indica al momento de ser creado y no puede cambiar. Por ejemplo, un canal de tipo `int` se define mediante la instrucción `make(chan int)`.

Si la capacidad de un canal es mayor a cero, quiere decir que hay un *buffer* asociado a

ese canal donde se guardarán los mensajes que esperan ser recibidos por algún proceso. Si no, se fuerza al canal a ser sincrónico: ejecutar una operación sobre él forzará a la goroutina a dormir hasta que otra goroutina ejecute la operación de sincronización correspondiente. Si existe un *buffer*, el canal puede funcionar de manera asincrónica en dos casos: si no está lleno y se ejecuta una operación de enviar un mensaje al canal y si no está vacío y se ejecuta una operación de recibir un mensaje del canal. En estos casos, la goroutina puede ejecutar la operación y luego continuar con su ejecución. Si en cambio, el *buffer* está vacío y se trata de recibir un mensaje de él o si el buffer está lleno y se trata de enviar, la goroutina se pondrá a dormir hasta que otra goroutina ejecute la acción de sincronización complementaria correspondiente. El comportamiento de la comunicación, por lo tanto, puede ser sincrónico o asincrónico según cada canal, el valor del buffer asociado a éste y la cantidad de elementos existentes en el *buffer*.

Los canales pueden cerrarse para indicar que ya no es posible enviar un mensaje a través suyo. Cualquier *goroutina* que tenga acceso a la referencia al canal puede cerrarlo. Si se intenta enviar un mensaje a través de un canal cerrado, se producirá una *panic failure* y se detendrá la ejecución del programa junto con todas las *goroutinas*. Lo mismo sucederá si se intenta cerrar un canal que ya está cerrado. Es posible, en cambio, en estos casos, recibir un mensaje siempre. Si el *buffer* del canal tenía elementos al momento de cerrarse, se reciben esos mensajes. Caso contrario, se recibe el valor *default* del tipo de datos del canal (0 para los *ints*, el *string* vacío para los *strings*, *false* para los booleanos, etc).

En la figura 2.1 podemos observar un ejemplo sencillo de comunicación sobre canales en Go donde la rutina que ejecuta el código de la función `main` crea el canal `ch` (linea 4) para transmitir valores enteros, luego declara la variable `dummy` con valor inicial 1 (linea 5), crea una gorutina que ejecuta la función `recv` (linea 6) y finalmente inicia un ciclo infinito en el que continuamente envía el valor 1 sobre el canal ch (lineas 7 - 9). La función `recv` recibe como parámetros un canal `ch` sobre el que se transmiten valores enteros y un entero `dummy` (linea 12). Esta función ejecuta un ciclo infinito en el que dependiendo del valor `dummy` recibido (línea 14) o envía el valor 1 sobre ch (línea 15) o no realiza ninguna acción. Como no se inicializa ningún *buffer* para el canal creado, la comunicación es sincrónica.

### 2.1.1. Compartir memoria comunicando

Go se caracteriza por su premisa de no comunicarse compartiendo memoria sino compartir memoria a través de la comunicación [11]. Su modelo de comunicación está basado (aunque no completamente) en el cálculo CSP (Comunicating Secuential Processes) de Hoare [12]. También incorpora elementos del Π-Calculus [13] como la capacidad de comunicar a través de canales entre más de un proceso paralelo. Entre otras razones, modelar la comunicación de esta forma logra un código mucho más sencillo y menos propenso a errores para ciertos tipos específicos de comunicación entre procesos como es el caso del modelo de productor-consumidor. También resultan útiles para modelar patrones de comunicación propios de la arquitectura de cliente-servidor que el propio servidor de la librería estandar de Go usa, como por ejemplo el Context que modela las interacciones que puede tener con las goroutinas que atienden un request (avisar que ya atendió el pedido, comunicar un error o informar de un *time-out* mientras aún estaba atendiendo), entre otras cosas. Además, los canales permiten agregar una noción de temporalidad al acceso al recurso compartido de una manera directa entre los procesos involucrados, y no

```
1. package main
2.
3. func main() {
4.         ch := make(chan int)
5.         dummy := 1
6.         go recv(ch, dummy)
7.         for {
8.                 ch <- 1
9.         }
10. }
11.
12. func recv(ch chan int, dummy int) {
13.         for {
14.                 if dummy == 1 {
15.                         <-ch
16.                 }
17.         }
18. }
```

*Fig. 2.1:* Ejemplo código en Go que envía y recibe infinitamente sobre un canal

mediados por una estructura auxiliar como podría ser un lock.

Hay tres particularidades que distinguen a Go respecto a su modelo de comunicación sobre canales:

- Colas FIFO: los canales se implementan mediante colas de prioridad FIFO con un tamaño predefinido al momento de la creación del canal que no puede ser modificado. El tamaño del *buffer* del canal estipula si la comunicación es sincrónica o asincrónica.

- *Goroutinas*: Go soporta threads livianos que se lanzan mediante el comando "go" y ejecutan en paralelo sin necesidad de realizar llamadas al sistema operativo. La nueva *goroutina* lanzada ejecuta la función referenciada luego de este comando.

- `Select`: `Select` en Go permite realizar una elección no determinística con una acción de comunicación en cada guarda. `Select` ejecuta la primer acción que puede tan pronto como hay disponible otra acción de comunicación con la que puede sincronizar. Si hay más de una acción disponible al mismo tiempo se ejecuta una al azar.

En [8] se propone utilizar una abstracción basada en el CCS de Milner [14] (Calculus of Comunicating Systems, que se diferencia de CSP, por ejemplo, por no distinguir entre una elección determinística y otra no determinística) combinando los fundamentos de ambos cálculos para aumentar la noción de tipo estricta de los canales de Go incorporando los conceptos de *liveness* y *safety*.

## 2.2. Session Types

Los tipos permiten clasificar entidades dentro de un programa para describir cuáles son las operaciones permitidas de una computación. Una disciplina de tipos puede garantizar que un programa bien tipado se comporta bien sin necesidad de ejecutarlo. En general, el enfoque de los sistemas de tipo está puesto en *cuál* debe ser el resultado de una computación.

Sin embargo, durante los años 90 surgieron un conjunto de trabajos que inspiraron nociones de tipado que son capaces de describir propiedades asociadas con el comportamiento de los programas [15]. Es decir, con *cómo* se debe realizar una computación. Ejemplos de esto son sistemas de tipados que permiten establecer una cota en la cantidad de memoria utilizada o sistemas de tipados que permiten asegurar que no se devolverá nunca *"message not understood"* en un lenguaje de programación orientado a objetos [15]. A los sistemas de tipado de estas características se les dió el nombre de *behavioural types*[1].

Varios de estos trabajos fueron desarrollando sistemas de tipado con características que luego serían utilizadas para el desarrollo de los tipos sesión. Los **Typestates** en el año 1986 [16] introducen una técnica para detectar invocaciones a métodos sintácticamente válidos pero semánticamente indeterminados como por ejemplo, acceder a una variable que no fue inicializada o acceder a un puntero que apunta a una estructura que fue liberada de la memoria. También permiten al compilador liberar la memoria de algún dato en el momento adecuado de una ejecución garantizando que el dato no se volverá a utilizar sin necesidad de un *garbage collector*. La principal novedad de los **Typestates** es sin embargo, el modelar un tipo como un estado que puede transicionar a un conjunto de otros estados posibles. Es decir, modela el tipo como una máquina de estados finita con transiciones (*Labeled Transitions System*).

Nierstrasz en 1995 propuso un sistema de tipado para el cálculo de objetos que permite identificar posibles invocaciones a mensajes no definidos en algún objeto ("*message not understood*") mediante la verificación de la propiedad de disponibilidad. Una de las principales novedades del trabajo de Nierstranz es el modelado de la composición en paralelo (en este caso de objetos pero entendidos como procesos autónomos) como un tipo. Kobayashi en el 2000, a partir de interpretar los procesos como tipos y a un tipo como la abstracción del comportamiento de ese proceso, desarrolla un sistema de tipado para un fragmento del $\pi$-calculus que permite verificar la ausencia de *deadlocks* y *livelocks* [17].

Los *Session Types* (tipos sesión) surgen como un sistema de tipado utilizado para modelar la programación basada en comunicación estructurada. Un tipo sesión tiene como propósito codificar la estructura correcta de comunicación entre distintos procesos para poder identificar comportamientos inadecuados sin necesidad de ejecutar las acciones de comunicación en sí. Los tipos sesión han tenido un gran desarrollo en los últimos años y hoy cuentan con implementaciones para distintos lenguajes de programación como C, Erlang, Go, Haskell, Java, OCaml, Python y Rust [18]. Como cada lenguaje de programación define su propio modelo de comunicación, los tipos que modelan cada uno también son distintos. En particular, el presente trabajo abarca el tipo sesión propuesto para la comunicación

[1] Tipos comportamentales

por canales en el lenguaje Go en [8].

### 2.2.1. Liveness y Safety

Junto con el desarrollo de los tipo sesión se han definido distintos conjuntos de propiedades que se desean verificar en un programa mediante su verificación en su tipo correspondiente. La propiedad de **liveness** de un programa se define como la habilidad de las acciones de comunicación de todos sus procesos de eventualmente ejecutarse. La propiedad no se refiere únicamente al progreso del programa en general sino específicamente al de todos los hilos de ejecución (podría existir un subconjunto de rutinas que siempre progresen mientras que una o más no lo hacen), es decir, a la ausencia de *deadlocks* tanto totales como parciales. La propiedad de **safety** se refiere a la ausencia de errores durante la ejecución. En el contexto de la verificación de programas en Go, se refiere particularmente a la semántica de sus canales que estipula que cerrar un determinado canal dos veces o ejecutar una acción de envío de un mensaje sobre un canal cerrado genera un error en tiempo de ejecución.

## 2.3. Fencing Off Go

En [8] se propone un lenguaje llamado MiGo que abstrae el comportamiento comunicacional de un programa Go. Luego, se introduce un sistema de tipos session para programas escritos en MiGo. Los tipos utilizados tienen una semántica operacional y por lo tanto pueden ejecutarse (simbólicamente). Las propiedades de un programa MiGo se analizan a partir del comportamiento de su tipo. Para ello, es esencial que se considere sólo un número finito de ejecuciones finitas de los tipos (que pueden ser recursivos). El trabajo considera, entonces, unicamente a los tipos *fenced*. Un tipo es *fenced* si está compuesto por una cantidad finita de patrones de comunicación y cualquier canal es utilizado por una cantidad finita de hilos de ejecución. Si cumple con esta propiedad, la ejecución simbólica es finita y termina, sin importar si el programa tiene ejecuciones infintas o no. Si la ejecución simbólica es finita, se pueden utilizar barbas sobre procesos para verificar si el programa cumple con las propiedades de **liveness** y **channel-safety**.

### 2.3.1. MiGo

La sintaxis de MiGo como cálculo basado en procesos CCS se puede observar en la figura 2.2. En términos generales, la sintaxis permite expresar tanto un comportamiento secuencial de un proceso, ordenando las ejecuciones mediante el intercalado del ";" como la composición en paralelo de dos procesos separándolos con "|". Permite tanto la elección determinística (if) como no determinística (select), crear y cerrar canales, ejecutar funciones (X$< \tilde{e}, \tilde{u} >$), declarar el nombre de un canal en un proceso (($\nu$c)P) y representar la cola de elementos v (todos de tipo $\sigma$) que hay en un canal c($\sigma$). A la vez permite expresar que un programa puede estar asociado a ciertas definiciones de funciones D.

Tomando el ejemplo de la figura 2.1, podemos observar en la figura 2.3 la implementación del mismo programa en lenguaje MiGo. El programa se define como la definición de dos funciones (S y R) dentro de un proceso que crea un canal y las invoca a ambas en paralelo. La funcion S, a su vez, envía sobre el canal tomado por parámetro y se llama

```
P,Q := π;P                          u := a | x
     | close u; P                   π := u!<e> | u?(y) | τ
     | select{π_i;P_i}_{i∈I}        v := n | true | false | x
     | if e then P else Q           e := v | not(e) | succ(e)
     | newchan(y:σ);P               D := X(x̃) = P
     | P | Q | 0                    P := {D_i}_{i∈I} in P
     | X<ẽ,ũ>                       σ := bool | int | ...
     | (νc)P
     | c<σ>::ṽ | c*<σ>::ṽ
```

*Fig. 2.2:* Sintaxis de MiGo según cómo se desarrolla en [8]

```
S(c)    ≜  c!⟨1⟩;S⟨c⟩
R(c,n)  ≜  if (n==0) then c?⟨x⟩;R⟨c,n⟩ else R⟨c,n⟩

{S(c),R(c,n)} in newchan(c:int);(S⟨c⟩|R⟨c,1⟩)
```

*Fig. 2.3:* Ejemplo de la figura 2.1 en lenguaje MiGo

recursivamente, mientras que la función R chequea si el parámetro `n` es igual a 0 y escucha del canal tomado como parámetro para luego llamarse recursivamente si es o sólo se llama recursivamente si no.

### 2.3.2. Modelo del Tipo Sesión

A continuación presentamos el modelo del tipo sesión introducido en el trabajo *Fencing Off Go* [8] sobre el que se basa el desarrollo de las herramientas **Dingo-Hunter** y **Gong** estudiadas[2]. Introducimos primero las reglas de tipado para los programas MiGo en la sección 2.3.2.1. Luego desarrollamos la semántica de los tipos en la sección 2.3.2.2 y finalmente mostramos la definición formal de **liveness** y **safety** para los programas MiGo utilizando barbas en la sección 2.3.2.3.

#### 2.3.2.1. Reglas de tipado para los programas MiGo

Las reglas de la figura 2.4 asignan tipos válidos a los programas MiGo para un proceso. Las reglas $< in >$ y $< out >$ estipulan que el envío o recepción de un mensaje e a través de un canal u seguidos de la ejecución de **P** tienen tipo $u!\langle e\rangle;T$ y $u?\langle y\rangle;T$ respectivamente siempre que u sea un canal que soporta mensajes de tipo $\sigma$ y el mensaje e enviado (o la variable y sobre la que se recive el mensaje respectivamente) sean también de tipo $\sigma$. Además, se requiere que la continuación P sea bien tipada, de tipo T. Para una acción de tipo $\tau$ seguida de la ejecución de **P** se asigna un tipo $\tau$ seguido del tipo **T** de **P**. Una acción de `close` de un canal u seguido del programa **P** tiene el tipo `end[u]` seguido

[2] En este trabajo sólo presentamos el modelo que soporta comunicación sincrónica. En la presentacion original del trabajo [8] el modelo se extiende para soportar comunicación asincrónica. Dado que el resto de este trabajo no depende de los detalles específicos de esta extensión, remitimos al lector interesado a ver detalles en la sección 6. Bounded Asynchrony in MiGo de la publicación mencionada.

del tipo **T** de **P**.

A la primitiva `select` que puede ejecutar un conjunto de acciones de comunicación $\pi_i$ seguidas cada una de un proceso $P_i$ se le asigna un tipo $\&\{\kappa_i; T_i\}$ donde cada $\kappa_i$ es el tipo que puede tomar la acción $\pi_i$ y cada $T_i$ es el tipo del $P_i$ correspondiente. La elección determinística `if`, que dependiendo de la guarda puede continuar con un proceso P o un proceso Q, genera el tipo $\oplus\{S, T\}$ siempre y cuando S sea el tipo de P y T el tipo de Q.

La creación de un nuevo canal referenciado por la variable **y** de tipo $\sigma$ seguida del proceso P genera el tipo $(new\ c)T\{^c/_y\}$ donde `c` es el nuevo canal creado referenciado por $y$ y T es el tipo de P asumiendo que $y$ es un canal de tipo $\sigma$. Notar que $c$ debe ser un nombre fresco. El tipo de la composición en paralelo de dos procesos es la composición en paralelo del tipo de cada proceso. La invocación de una función X tiene tipo siempre y cuando esté definida en el contexto con parámetros de tipo $\tilde{\sigma}$ para las expresiones y $ch(\tilde{\sigma}')$ para los canales y a los parámetros que se le pasen se les asigne los tipos adecuados con ese mismo contexto.

En la figura 2.5 tenemos también las reglas de tipado para un programa en tiempo de compilación y durante la ejecución (`runtime`). Un programa consta de un proceso principal asociado a un conjunto de definiciones de funciones y se le puede asignar un tipo estáticamente siempre y cuando todas las definiciones de funciones tengan tipo y el proceso principal también tenga tipo al agregar esas definiciones al contexto. Las reglas de tipado dinámicas establecen los tipos de los `buffers` y su asociación a un canal existente. Para definirlas, se agrega un nuevo juicio $\Gamma \vdash_B P \blacktriangleright T$ que implica la asignación de un tipo T a un proceso P en un contexto $\Gamma$ con un conjunto B de *buffers* creados en tiempo de ejecución.

La regla $< res >$ estipula que un proceso $(\nu c)$P que define un nuevo canal c tiene tipo $(\nu c)$T bajo un contexto $\Gamma$ con un conjunto de *buffers* **B-{c}** activos en la ejecución, siempre que P tenga tipo T bajo el contexto $\Gamma$ extendido con el tipo $\sigma$ del canal c y con un conjunto **B** de *buffers* activos. Los *buffers* abiertos $(a\langle\sigma\rangle)$ y cerrados $(a^*\langle\sigma\rangle)$ serán de tipo $\lfloor \mathbf{a} \rfloor$ y $\mathbf{a}^*$ respectivamente si el canal al que están asociados es del tipo correspondiente $(ch(\sigma))$ y el *buffer* a fue definido previamente en la ejecución $(\vdash_a)$. Por último, la regla $< parr >$ estipula que la composición en paralelo de dos procesos tiene tipo bien formado con un conjunto de *buffers* si uno de los procesos tiene tipo bien formado con un subconjunto de esos *buffers* mientras que el otro tiene tipo válido con el complemento de ese subconjunto.

#### 2.3.2.2. Semántica de tipos

Para todo tipo que representa el comportamiento comunicacional de un programa existe un *Labeled Transition System* (de ahora en adelante LTS) que representa su ejecución. Para los tipos que cumplan con la propiedad de ser *fenced*, se puede garantizar que el LTS que representa su ejecución tiene una cantidad de estados *finita*. Por esta razón es que pueden ser ejecutados simbólicamente para constatar las propiedades de **liveness** y **safety**. El LTS asociado a un tipo está definido inductivamente por las reglas de inferencia que se muestran en la figura 2.6.

Las etiquetas de las transiciones representan las posibles acciones que el tipo puede tomar y se definen con las letras $\alpha$ y $\beta$. Las posibles acciones son $\bar{a}$, que representa el

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

$$<out>\ \frac{\Gamma \vdash \text{u:ch}(\sigma)\quad \Gamma \vdash \text{e:}\sigma\quad \Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \text{u!}\langle\text{e}\rangle;\ \text{P} \blacktriangleright \bar{u};\ \text{T}} \qquad <sel>\ \frac{}{\Gamma \vdash \pi_i; P_i \blacktriangleright \kappa_i; T_i}$$

$$\Gamma \vdash \text{select}\{\pi_i; P_i\}_{i\in I} \blacktriangleright \&\{\kappa_i; T_i\}_{i\in I}$$

$$<in>\ \frac{\Gamma \vdash \text{u:ch}(\sigma)\quad \Gamma,\ \text{x:}\sigma \vdash P \blacktriangleright T}{\Gamma \vdash \text{u?(x);}\ \text{P} \blacktriangleright \text{u;}\ \text{T}} \qquad <if>\ \frac{\Gamma \vdash e : bool\quad \Gamma \vdash P \blacktriangleright S\quad \Gamma \vdash Q \blacktriangleright T}{\Gamma \vdash \text{if e then P else Q} \blacktriangleright \oplus\{\text{S, T}\}}$$

$$<tau>\ \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \tau;\ \text{P} \blacktriangleright \tau;\ \text{T}} \qquad <new>\ \frac{\Gamma, y : ch(\sigma) \vdash P \blacktriangleright T\quad \text{c} \notin \text{dom}(\Gamma)\cup\text{fn(T)}}{\Gamma \vdash \text{newchan(y:}\sigma\text{);P} \blacktriangleright \text{(new c)T}\{^{\text{c}}/_{y}\}}$$

$$<close>\ \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash \text{close u;}\ \text{P} \blacktriangleright \text{end[u];}\ \text{T}} \qquad <par>\ \frac{\Gamma \vdash P \blacktriangleright T\quad \Gamma \vdash Q \blacktriangleright S}{\Gamma \vdash \text{P} \mid \text{Q} \blacktriangleright \text{T} \mid \text{S}}$$

$$<zero>\ \frac{}{\Gamma \vdash 0 \blacktriangleright 0} \qquad <var>\ \frac{\Gamma \vdash \tilde{e} : \tilde{\sigma}\quad \Gamma \vdash \tilde{u} : ch(\tilde{\sigma'})}{\Gamma,\text{X}(\tilde{\sigma}, ch(\tilde{\sigma'})) \vdash \text{X}\langle\tilde{e}, \tilde{u}\rangle\ \blacktriangleright t_x\langle\tilde{u}\rangle}$$

*Fig. 2.4:* Reglas de tipado para un proceso

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

$$<def>\ \frac{\begin{array}{c}\Gamma, X_i(\tilde{\sigma}_i, ch(\tilde{\sigma}_i)), \tilde{x}_i : \tilde{\sigma}'_i, \tilde{y} : ch(\tilde{\sigma}'_i) \vdash P_i \blacktriangleright T_i \\ \Gamma, X_1(\tilde{\sigma}_1, ch(\tilde{\sigma}'_1)), ..., X_n(\tilde{\sigma}_n, ch(\tilde{\sigma}'_n)) \vdash Q \blacktriangleright S\end{array}}{\Gamma \vdash \{X_i(\tilde{x}_i, \tilde{y}_i) = P_i\}_{i\in I}\ in\ Q \blacktriangleright \{t_{X_i}(\tilde{y}_i) = T_i\}_{i\in I}\ in\ S}$$

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

$$<int>\ \frac{\Gamma \vdash P \blacktriangleright T}{\Gamma \vdash_\emptyset P \blacktriangleright T} \qquad <buff>\ \frac{\Gamma \vdash a : ch(\sigma)}{\Gamma \vdash_{\{a\}} a\langle\sigma\rangle :: \tilde{v} \blacktriangleright \lfloor a \rfloor}$$

$$<res>\ \frac{\Gamma, c : ch(\sigma) \vdash_B P \blacktriangleright T}{\Gamma \vdash_{B\setminus c} (\nu c)P \blacktriangleright (\nu c)T} \qquad <parr>\ \frac{\Gamma \vdash_B P \blacktriangleright T\quad \Gamma \vdash_{B'} Q \blacktriangleright S\quad B \cap B' = \emptyset}{\Gamma \vdash_{B\cup B'} P|Q \blacktriangleright T|S}$$

$$<cbuff>\ \frac{\Gamma \vdash a : ch(\sigma)}{\Gamma \vdash_{\{a\}} a^*\langle\sigma\rangle :: \tilde{v} \blacktriangleright a^*}$$

*Fig. 2.5:* Reglas de tipado estáticas y dinámicas (`runtime`) para un programa

envío de un mensaje a través del canal a; $a$, que representa el recibir un mensaje del canal a; $\tau$ que representa una acción silenciosa (no tiene efectos en cuanto al comportamiento comunicacional y siempre puede ejecutarse, por ejemplo, un *timeout*); $[a]$ que representa la acción de sincronización sobre el canal a; $end[a]$ que representa la acción de solicitar el cierre de un canal; $\overline{end}[a]$ que representa la respuesta al pedido de cierre de un canal y $a^*$ que representa el envío del valor *default* de un canal cerrado.

Las reglas $[snd]$ y $[rcv]$ permiten a un tipo enviar y recibir simbolicamente un mensaje a través del canal $a$. La regla $[tau]$ permite continuar la ejecución luego de una acción de tipo $\tau$. La regla $[sel]$ estipula para los casos de elecciones internas que cualquiera de las acciones contenidas en la elección puede ejecutarse mediante una reducción de tipo $\tau$. La regla $[bra]$ estipula que para las elecciones externas se puede reducir a $T_j$ mediante una acción $\alpha$ siempre y cuando el término $\kappa_j; T_j$ contenindo en la elección reduzca a $T_j$ mediante una acción $\alpha$. La regla $[par]$ estipula que la composición de dos tipos en paralelo puede ejecutar cualquier acción que pueda realizar cualquiera de los tipos de la composición. Notar que esta regla puede aplicar para cualquiera de los tipos de la composición debido a la regla $[eq]$, que estipula que un tipo puede reducir a cualquier cosa a la que reduzca otro tipo que sea $\alpha$-equivalente consigo. Las reglas de congruencia necesarias para definir la relación de $\alpha$-equivalente pueden observarse en la parte inferior de la figura.

La regla $[com]$ estipula que para poder realizar una acción de sincronización sobre una composición en paralelo de tipos, uno de ellos tiene que reducir mediante una acción de recibir $(a)$ y el otro tiene que reducir mediante una acción o bien de enviar $(\bar{a})$ o bien de enviar el valor *default* desde un canal cerrado ($a^*$). La regla $[new]$ define que desde un estado donde se crea un nuevo canal previo al tipo **T**, se transiciona mediante una acción silenciosa $(\tau)$ a una composición en paralelo de **T** con un buffer abierto para el canal, donde en ambos se referencia al canal creado con una variable ($(\nu a)$). Las reglas $[res-1]$ y $[res-2]$ son las que definen el comportamiento general de la ejecución semantica ya que estipulan que donde hay una referencia a un canal dentro del tipo **T** ($(\nu a)$T), cualquier acción que no involucre al canal reduce a lo que reduciría **T** al ejecutar esa acción manteniendo la definición del canal ($(\nu a)$T') y cualquier acción que involucre al canal reduce mediante una acción $\tau$ a ($(\nu a)$T') sólo si T puede reducir mediante una acción de sincronización sobre $a$ ([a]) a T'. Estas reglas, en última instancia son las que garantizan que la reducción semántica se realizará respetando la semántica de Go mediante la cual no se puede reducir la acción de enviar por un canal si en paralelo no existe otro proceso que recibe de ese canal o viceversa.

Las reglas $[end]$, $[buf]$ y $[close]$ definen el comportamiento de cierre de un canal. La regla $[end]$ establece que para un tipo compuesto por $end[a]$ seguido de **T** se transiciona a **T** mediante la acción $end[a]$. La regla $[buf]$ establece que de un buffer de un canal abierto se transiciona a un buffer de ese canal pero cerrado mediante la acción $\overline{end}[a]$. Luego $[close]$ define como estas dos reglas se usan de conjunto para sincronizar el cierre del canal entre dos procesos en paralelo, el que lo solicita y el del buffer, que responde cerrándose. Si el canal está definido, la regla $[res-2]$ restringe la aplicación de $[end]$ y $[buf]$ de manera aislada ya que son acciones realizadas sobre un canal definido. Si existen dos procesos en paralelo, uno capaz de cerrar y otro capaz de responder al cierre (un buffer abierto) la regla $[close]$ puede aplicarse en combinación con $[res-1]$ debido a que la acción definida

$$[snd]\ \bar{a};T \xrightarrow{\bar{a}} T \qquad [end]\ end[a];T \xrightarrow{end[a]} T$$

$$[rcv]\ a;T \xrightarrow{a} T \qquad [buf]\ \lfloor a \rfloor \xrightarrow{end[a]} a^*$$

$$[tau]\ \tau;T \xrightarrow{\tau} T \qquad [close]\ \frac{T \xrightarrow{end[a]} T' \quad S \xrightarrow{\overline{end}[a]} S'}{T \mid S \xrightarrow{\tau} T' \mid S'}$$

$$[sel]\ \frac{j \in I}{\oplus\{T_i\}_{i\in I} \xrightarrow{\tau} T_j} \qquad [res{-}1]\ \frac{T \xrightarrow{\alpha} T' \quad fn(\alpha) \neq \{a\}}{(va)T \xrightarrow{\alpha} (va)T'}$$

$$[bra]\ \frac{\kappa_j;T_j \xrightarrow{\alpha} T_j}{\&\{\kappa_i;T_i\}_{i\in I} \xrightarrow{a} T_j} \qquad [res{-}2]\ \frac{T \xrightarrow{[a]} T'}{(va)T \xrightarrow{\tau} (va)T'}$$

$$[par]\ \frac{T \xrightarrow{\alpha} T'}{T \mid S \xrightarrow{\alpha} T' \mid S} \qquad [eq]\ \frac{T \equiv_\alpha T' \quad T \xrightarrow{\alpha} T''}{T' \xrightarrow{\alpha} T''}$$

$$[com]\ \frac{T \xrightarrow{\beta} T' \quad S \xrightarrow{a} S' \quad \beta = \bar{a}, a^*}{T \mid S \xrightarrow{[a]} T' \mid S'} \qquad [def]\ \frac{T\{\tilde{\alpha}/\tilde{x}\} \xrightarrow{\alpha} T' \ \mathbf{t}(\tilde{x}) = T}{\mathbf{t}\langle\tilde{\alpha}\rangle \xrightarrow{\alpha} T''}$$

$$[new]\ (new\ a)T \xrightarrow{\tau} (va)(T \mid \lfloor a \rfloor)$$

$$\alpha, \beta\ ::=\ \bar{a} \mid a \mid \tau \mid [a] \mid end[a] \mid \overline{end}[a] \mid a^*$$

$$T \mid S \equiv S \mid T \quad T \mid (T' \mid S) \equiv (T \mid T')\ S \quad T \mid 0 \equiv T$$
$$(va)(vb)T \equiv (vb)(va)T \quad (va)0 \equiv 0 \quad (va)a^* \equiv 0 \quad (va)\lfloor a \rfloor \equiv 0$$
$$T \mid (va)S \equiv (va)(T \mid S)_{a\notin fn(T)} \quad T \equiv_\alpha T' \Rightarrow T \equiv T'$$

*Fig. 2.6:* Semántica operacional para la ejecución simbólica de los tipos tomada de [8]

es de tipo $\tau$ y no se considera una acción que involucre el nombre del canal que se está cerrando.

Por último la regla $[def]$ estipula que el tipo de la invocación a una función puede transicionar a lo mismo a lo que transiciona el tipo del programa referenciado por esa invocación reemplazando las referencias correspondientes por referencias a los canales pasados por parámetro.

#### 2.3.2.3. Liveness y Safety

Para definir formalmente los conceptos de **Liveness** y **Safety** en el contexto de MiGo se utilizan barbas. Las barbas son predicados atómicos que pueden satisfacerse o no en una determinada reducción semántica y tienen un antecedente en el concepto de *observable básico* de una reducción. Junto con la semántica de reducción, permiten establecer distintos conceptos de equivalencia entre procesos [19].

En la figura 2.7 podemos observar las barbas definidas para la reducción semántica de la figura 2.6. En términos generales, $\downarrow$ es un predicado que representa la existencia de una acción de comunicación. Más específicamente, $\downarrow_a$ hace referencia a la acción de intentar recibir un mensaje del canal $a$, $\downarrow_{\bar{a}}$ hace referencia a la acción de intentar enviar un mensaje por el canal $a$, $\downarrow_{end[a]}$ hace referencia a la acción de cerrar un canal, $\downarrow_{a*}$ hace referencia a la posibilidad de enviar el valor default del canal cerrado $a$ cuando se

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

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

$$\frac{Q\{^{\bar{e},\bar{a}}/_{\bar{x}}\}\downarrow_o \quad X(\bar{x})=Q}{X\langle\bar{e},\bar{a}\rangle\downarrow_o} \qquad \frac{\forall i\in\{1,...,n\}\ :\ \pi_i\downarrow_{o_i}}{select\{\pi_i; P_i\}_{i\in\{1,...,n\}}\downarrow_{o_1,...,o_n}} \qquad \frac{P\downarrow_a \quad Q\downarrow_{\bar{a}}\ or\ Q\downarrow_{a^*}}{P \mid Q\downarrow_{[a]}}$$

$$\frac{P\downarrow_a \quad \pi_i\downarrow_{\bar{a}}}{P \mid select\{\pi_i; P_i\}_{i\in I}\downarrow_{[a]}} \qquad \frac{P\downarrow_{\bar{a}}\ or\ P\downarrow_{a^*} \quad \pi_i\downarrow_a}{P \mid select\{\pi_i; P_i\}_{i\in I}\downarrow_{[a]}}$$

$$P\Downarrow_o\ \ if\ P\rightarrow^* P'\ and\ P'\downarrow_o\ \ with\ o\in\{c,\bar{c},[c],end[c],c^*\}$$

*Fig. 2.7:* Predicados de las barbas para Liveness y Safety

trata de recibir un mensaje de él y $\downarrow_{[a]}$ hace referencia a la acción de sincronización de dos acciones de comunicación. Este último caso sólo está definido para dos procesos en paralelo para los cuales uno satisface el predicado $\downarrow_a$ y el otro satisface o bien $\downarrow_{\bar{a}}$ o bien $\downarrow_{a*}$.

Si un programa satisface un determinado predicado, ese predicado se satisface para la composición tanto secuencial como en paralelo de ese programa con cualquier otro. Una invocación a un programa satisface los predicados del programa invocado, reemplazando las variables correspondientes por las que se le pasan por parámetro a la invocación. Si dos programas son equivalentes entre sí, satisfacen los mismos predicados. Por último, para el caso del `select`, un `select` satisface potencialmente todos los predicados de cualquiera de las acciones posibles que puede ejecutar. Esto se representa concatenando las acciones posibles luego de $\downarrow$. Por ejemplo, $\downarrow_{b,\bar{a}}$ representa la posibilidad de o bien recibir de b o bien enviar por a. A la vez, genera un predicado de sincronización si puede sincronizar una de sus acciones posibles con la acción de una rutina paralela.

Utilizando las barbas propuestas, se puede definir formalmente los predicados de **liveness** y **safety** en términos de relaciones entre predicados que deben o no cumplirse. La definición formal de las propiedades puede verse en la figura 2.8. Un programa **P** satisface **liveness** si al reducir a un programa ($\nu$c)Q cumple que si Q satisface el predicado de haber enviado o recibido sobre c entonces eventualmente va a satisfacer el predicado de haber sincronizado esa acción. A la vez, también cumple que si Q satisface un predicado con un conjunto de posibles acciones de comunicación, eventualmente va a satisfacer la sincronización de alguna de esas acciones. Un programa **P** satisface **safety** si al reducir a un programa ($\nu$c)Q se cumple que si Q satisface la propiedad de tener el canal a cerrado entonces en ningún punto de la ejecución futura de Q se va a satisfacer la propiedad de enviar sobre ese canal o de volver a requerir el cierre del mismo.

## 2.4. Implementación

La implementación consiste de dos herramientas. **Dingo-Hunter**, por un lado, toma como parámetro un código escrito en Go y devuelve el código MiGo correspondiente. **Gong** luego *parsea* el código MiGo, construye su tipo y chequea si cumple la propiedad

**Liveness:** El programa **P** satisface **liveness** si
$\forall$ Q tq **P** $\rightarrow^* (\nu c)Q \Rightarrow$
a) $Q \downarrow_a$ or $Q \downarrow_{\bar{a}} \Rightarrow Q \Downarrow_{[a]}$
b) $Q \downarrow_{\tilde{a}} \Rightarrow Q \Downarrow_{[a_i]}$ con $a_i \in \tilde{a}$

**Safety:** El programa **P** es seguro en su comunicación con canales si:
$\forall$ Q tq **P** $\rightarrow^* (\nu c)Q \Rightarrow$
if $Q \downarrow_{a^*}$ then $\neg(Q \Downarrow_{end[a]}) \wedge \neg(Q \Downarrow_{\bar{a}})$

*Fig. 2.8:* Definición de liveness y safety utilizando barbas


*Fig. 2.9:* Etapas de la extracción de un programa MiGo a partir de un código Go en **Dingo-Hunter**

de ser *fenced*. Si la cumple, lo ejecuta simbólicamente y contruye el LTS que representa los estados de ejecución del programa de acuerdo con las reglas descriptas en la figura 2.6. Un estado de ejecución es aquel en donde sólo se puede avanzar en la ejecución o bien mediante la sincronización de dos operaciones (un envío y una recepción sobre un canal, incluyendo las operaciones que puedan encontrarse dentro de un `select`, o un cierre y una confirmación del cierre) o bien mediante una elección interna (`if`). El LTS puede analizarse luego para verificar si todos los estados cumplen la propiedad de ser *live* y/o *safe*. Un programa es *live* o *safe* si todos sus estados son *live* o *safe* respectivamente. Gong devuelve un *booleano* por cada propiedad indicando si el programa la satisface o no.

Dentro del repositorio de **Dingo-Hunter** se puede encontrar, además, un servicio *web* simple que otorga una interfaz gráfica donde se puede escribir un código en Go (o cargar un conjunto de ejemplos predefinidos) y el servidor extrae primero el código MiGo y luego ejecuta Gong, devolviendo por la interfaz el resultado.

### 2.4.1. Dingo-Hunter

En la figura 2.9 se puede observar un esquema general de la herramienta **Dingo-Hunter** dividida en tres etapas. En la primer etapa, se toma un código en Go como parámetro y se lo *parsea* a una representación SSA intermedia que, entre otras cosas, simplifica la estructura de control de flujo del programa. En la segunda etapa se abstrae de esa representación el comportamiento comunicacional, filtrando las operaciones pertinentes y transformándolas en estructuras llamadas *statements* que se agrupan dentro de estructuras propias de MiGo llamadas `Functions`. En la última etapa, se extrae

```
def main.main():
    let t0 = newchan main.main.t0_0_0, 0;
    spawn main.recv(t0);
    call main.main#1(t0);
def main.main#1(t0):
    send t0;
    call main.main#1(t0, t0);
def main.recv(ch):
    call main.recv#1(ch);
def main.recv#1(ch):
    if recv ch; call main.recv#1(ch); else call main.recv#1(ch); endif;
```

*Fig. 2.10:* Resultado de ejecutar la herramienta dingo-hunter sobre el ejemplo de la figura 2.1

el código propiamente dicho del programa MiGo a partir de los *statements* y se lo devuelve.

En la figura 2.10 podemos ver el resultado de correr la herramienta sobre el ejemplo de la figura 2.1. Podemos observar algunas diferencias en cuanto al término de la figura 2.3 y el código. Mientras que el término utiliza expresiones básicas compuestas de *booleanos* y naturales, el código generado sólo abstrae las acciones estrictamente de comunicación. Por ese motivo ignora tanto la guarda del if de la línea 14 (la cual se representa como "`if P; else Q; endif`", sin condición de elección entre P y Q) como el segundo parámetro que toma la función Recv (que no usa dado que el if no tiene guarda). Un if, por lo tanto, si bien se distingue como elección interna, asume que cualquiera de los dos bloques de la elección podría ejecutarse siempre. A la vez, el código no distingue el tipo de un canal. Esto hace que sea imposible aplicar las reglas de tipado de la figura 2.4. Sin embargo, el generador de MiGo compila el código Go tomado por parámetro antes de generar el código MiGo y falla si el código original no compila utilizando el compilador estandar de Go. De esta manera se puede garantizar que si el código MiGo es generado por **Dingo-Hunter**, cumple automáticamente con la parte de las reglas correspondiente a invocar una acción sobre un canal con el tipo correspondiente.

#### 2.4.1.1. SSA

Para poder extraer el código MiGo de un código Go es necesario primero representar el código en una forma intermedia que simplifique su sintaxis y abstraiga su comportamiento para luego filtrar las acciones que sean específicas del comportamiento comunicacional. Con este objetivo se utiliza la representación SSA [20] que originalmente fue concebida para optimizar las decisiones tomadas por el compilador de distintos lenguajes de programación. Más allá de los cambios introducidos en el código de un programa por esta representación, es importante destacar a los efectos de este trabajo el impacto que tiene en las estructuras de control de flujo: los `for` y `while` se reemplazan por condicionales y `jumps` con los que se van a modelar los ciclos en el código MiGo extraído.

Observamos en la figura 2.11 el resultado de extraer la representación SSA del ejemplo de la figura 2.1. Extraemos la representación SSA mediante la herramienta **Dingo-Hunter** [9], la cual utiliza a su vez la librería ssa [21] de Go, pasándole como parámetro

```
# Name: main.main
# Package: main
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
        ; var dummy int @ 5:2 is 1:int
        ; func main.recv(ch chan int, dummy int) @ 6:5 is recv
        ; var ch chan int @ 6:10 is t0
        ; var dummy int @ 6:14 is 1:int
        go recv(t0, 1:int)
        jump 1
1:                                                          for.body P:2 S:1
        ; var ch chan int @ 8:3 is t0
        send t0 <- 1:int
        jump 1

# Name: main.recv
# Package: main
0:                                                             entry P:0 S:1
        jump 1
1:                                                          for.body P:3 S:2
        ; var dummy int @ 14:6 is dummy
        t0 = dummy == 1:int                                             bool
        ; *ast.BinaryExpr @ 14:6 is t0
        if t0 goto 2 else 1
2:                                                           if.then P:1 S:1
        ; var ch chan int @ 15:6 is ch
        t1 = <-ch                                                        int
        ; *ast.UnaryExpr @ 15:4 is t1
        jump 1
```

*Fig. 2.11:* Representación SSA del ejemplo de la figura 2.1

el comando `extractssa`. En el ejemplo podemos ver cómo el ciclo de la figura principal se representa con un `jump` a la etiqueta 1 mientras que en la función `Recv` el ciclo se modela con un condicional que chequea la condición y va a la etiqueta 2 (recibir del canal) si es verdadera o vuelve a la etiqueta 1 si no. El código MiGo generado a partir de esta representación heredará esta misma forma de representar el control del flujo.

#### 2.4.1.2. Generando los *Statements*

Una vez obtenida la representación SSA del programa, **Dingo-Hunter** construye un extractor de código Migo que filtra las instrucciones quedándose sólo con aquellas relacionadas al comportamiento comunicacional y genera *Statements* para cada una de ellas. Los *Statements* son estructuras que guardan datos pertinentes a cada instrucción que además se agrupan al compartir una misma interfaz. Las interfaces son parte del lenguaje Go y permiten asociar tipos de datos de acuerdo a si saben responder a un determinado conjunto de funciones. En particular, los *Statements* saben responder al método `String`, el cual devuelve la instrucción representada por el *Statement* en formato texto.

El extractor de código llama a la función `visitFunc` para los métodos Init y Main respectivamente y respetando ese orden. Dentro de ésta se llama, a su vez, a la función `visitBasicBlock` para el primer bloque de código y a su vez dentro de esta función se llama a `visitInstr` para cada una de las instrucciones del bloque. Ésta última función

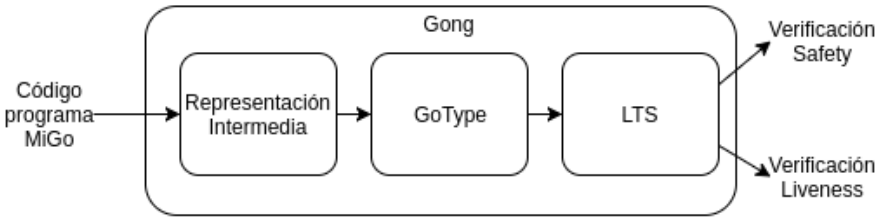


*Fig. 2.12:* Esquema de la herramienta **Gong**

es la que genera los *Statements* sólo para las instrucciones correspondientes (se saltea el resto) y los agrega al contexto de la función MiGo mediante la invocación a la función `AddStmts`. Finalmente se itera sobre los distintos tipos de *statements* utilizando la interfaz y llamando al método `String` para imprimir cada una de las líneas del código MiGo.

### 2.4.2. Gong

**Gong** es un programa desarrollado en Haskel que toma como parámetro de entrada un código escrito en MiGo y devuelve dos valores *booleanos* indicando para cada una de las propiedades de *liveness* y *safety* si puede garantizar que se cumple o no. En la figura 2.12 podemos observar el esquema general de la implementación de la herramienta.

En primer lugar, se *parsea* el texto de entrada con el fin de obtener una representación abstracta interna del programa MiGo. En esta etapa se valida que la sintaxis del código sea correcta y que el texto de entrada represente un programa MiGo válido. Luego, sobre esa representación intermedia, se infiere el tipo (`GoType`) que se define recursivamente y representa al programa en su estado inicial.

Teniendo el tipo del programa **Gong** construye un LTS que representa los distintos estados por los que puede pasar la ejecución de un programa. Este proceso se detalla en la sección 2.4.2.3. Por ultimo, se recorren los estados del LTS para verificar si cumplen las propiedades de *liveness* y *safety* definidas en la sección 2.3.2.3. Este proceso se detalla en las secciónes 2.4.3 y 2.4.4.

#### 2.4.2.1. Representación intermedia

El tipo de datos correspondiente a la representación intermedia utilizada por **Gong** para los programas MiGo válidos puede verse en la figura 2.13. **Gong** agrupa las instrucciones en una lista que preserva el orden en el que aparecen en el texto de entrada e interpreta el orden de esa lista como el orden de ejecución.

#### 2.4.2.2. GoTypes

Los tipos sesión utilizados se representan en Haskell con el tipo de datos algebraico `GoType` que se muestra en figura 2.14. Un tipo para un programa MiGo se represena mediante un valor de tipo algebráico `Eqn`. Intuitivamente un valor de tipo `Eqn` consiste de un conjunto de ecuaciones que definen al tipo de las funciones auxiliares (es decir, [(`EqnName`

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

*Fig. 2.13:* Representación intermedia del código MiGo parseado antes de que se realize el chequeo de tipos

```
type EqnName = Name GoType

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

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

*Fig. 2.14:* GoTypes definidos en la versión original de Gong

, Embed GoType)]) y un Gotype que representa el comportamiento del programa principal. Cada definición de funcion auxiliar (EqnName, Embed GoType) asocia un nombre (esto es, el nombre de la función) al tipo que describe su comportamiento. Es importante notar que estas funciones pueden ser mutuamente recursivas, es decir, los nombres EqnName pueden ser invocados tanto desde los GoTypes correspondientes a cualquiera de las funciones auxiliares como desde el del programa principal. Se utiliza la monada Bind para representar el hecho de que el nombre utilizado no es importante y se trabaja a menos de alfa-conversión.

La definición del tipo GoType se correspnde con la sintáxis de tipos descripta en la sección 2.3.2.1. Los generadores Send, Receive y Close tienen dos argumentos: el numbre del canal sobre el que se ejecutan y el GoType que representa a la continuacióon. El Tau toma solamente la continuación. Las elecciones internas y externas (IChoice y OChoice) se asocian a los conjuntos de GoTypes que representan las diferentes posibles ejecuciones de la elección. La composición en paralelo (Par) tiene una lista de GoTypes representando las distintas ejecuciones paralelas. New representa la creación de un canal: toma un entero representando el tamaño del buffer asociado y genera un nombre *fresh* para el canal creado que puede ser utilizado en la continuación. Esto se realiza utilizando la mónada Bind. Null representa la finalización de un hilo de ejecución.

TVar representa una invocación a una función y tiene un parámetro con el nombre (EqnName) de la función que está siendo invocada. ChanInst representa el pasaje como parámetros de un conjunto de canales a una función invocada y se construye con un GoType que sólo puede ser de tipo TVar, representando a la función y una lista de nombres de canales que representa a los parámetros que se le pasan. Seq representa la ejecución secuencial y toma una lista de GoTypes que representan una ejecución en orden según el orden de la lista. Esta instrucción es necesaria para modelar la invocación de una función seguida de algún otro tipo. **Gong**, si bien es capaz de construir el tipo para estos casos, al realizar la ejecución simbólica no acepta una instrucción Seq con más de un elemento. Es decir, asume que al realizar un Call, nunca se hará un retorno de la función llamada para seguir ejecutando otras instrucciones luego.

Por último los tipos Buffer y ClosedBuffer representan los buffers asociados a un determinado canal. Mientras que el segundo sólo toma el nombre del canal al cual está asociado, el primero toma además del nombre del canal, un *booleano* que indica si el canal está abierto o cerrado y dos enteros que indican la capacidad del buffer y la cantidad de elementos que tiene.

El chequeo de tipos se realiza en la función transformProg que toma como parámetro un programa representado como las definiciones de sus funciones auxiliares por un lado y la función principal por el otro, y para cada una encuentra su tipo (GoType) mediante la función transformSeq. Luego devuelve un tipo Eqn que asocia el tipo de las funciones auxiliares junto con sus nombres al tipo de la función principal.

La función transformSeq es la implementación recusiva de la definición inductiva del sistema de tipos de la figura 2.4. Ésta toma dos parámetros: un contexto $\Gamma$ (llamado vars) que se inicializa vacío, y una representación intermedia de la función que se busca tipar. La representación de la función consiste en una lista de tipo Interm (figura 2.13) que se

interpreta en los términos descriptos en la sección 2.4.2.1.

#### 2.4.2.3. Labeled Transition System (K-States)

Al construir el LTS que representa una ejecución simbólica acotada del programa la cuestión fundamental es cómo se define un estado. Los estados del LTS nos sirven para constatar en cada momento de la ejecución si se cumplen o no las propiedades que se desea verificar. La verificación de esas propiedades depende exclusivamente de los predicados generados por las barbas de la figura 2.7. En ese sentido, para definir un estado del LTS se dividen las instrucciones en dos tipos: las que generan guardas (`Send`, `Recv`, `Close`, `Buffer`, `Tau`, `IChoice` y `OChoice`) y las que no (`New`, `TVar`, `ChanInst`, `ChanAbst`, `Par` y `Seq`). Las guardas se generan siempre que sea necesario validar una barba que involucra una composición en paralelo (es decir, una sincronización). La idea de esta clasificación es que las instrucciones del primer grupo dependen de acciones de sincronización para progresar. Las acciones de sincronización pueden ser de tres tipos:

- La comunicación a través de un canal que involucra un hilo de ejecución que envía (`Send`) y otro que recibe (`Recv`). El tipo de la elección externa (`OChoice`) genera guardas que pueden validar cualquiera de estas acciones dependiendo de los tipos que tenga dentro
- El cierre de un canal, que involucra a un hilo de ejecución que realiza el pedido de cierre y un *buffer* que se cierra. En un programa Go sólo existe la rutina que ejecuta la instrucción `close`, pero el tipo modela la acción interna del compilador de Go en el cual el *buffer* de un canal tiene un estado paralelo con respecto a las rutinas que se ejecutan
- Una acción "silenciosa"de tipo `tau`. Esta acción no requiere en verdad que existan procesos en paralelo sino que realiza la sincronización por si sóla. Representa, por ejemplo, un temporizador que envía un mensaje automáticamente luedo de un determinado período de tiempo (`tau`). El tipo de una elección externa (`OChoice`) puede realizar esta acción si tiene un tipo `Tau` dentro. A su vez, las elecciones internas (`IChoice`) se modelan como una elección externa entre dos acciones silenciosas, por lo que realizan también este tipo de sincronización.

Los estados de un LTS son entonces los tipos que representan el momento de una computación inmediatamente previo a realizar una sincronización y cada uno tiene una arista etiquetada con un tipo de sincronización específico si ejecutando únicamente esa acción de sincronización se llega al momento de ejecución representado por el tipo del estado siguiente.

El LTS nos permite verificar propiedades a lo largo de la ejecución garantizando que si en algún momento se vuelve a un estado ya verificado no es necesario verificarlo de nuevo. Esto le permite analizar programas que corren infinitamente.

En la figura 2.15 se observan los dos tipos que representan los dos estados de la LTS generada para el ejemplo de la figura 2.1. En el estado 0, los tres tipos compuestos en paralelo son el resultado de realizar las siguientes acciones:

- crear el canal de la línea 4 (`Buffer`)

```
Estado 0:

(New (-1) (bind t06 (
    Par [
        IChoice
            (Recv 0@0 (ChanInst (TVar 1@2) [0@0]))
            (ChanInst (TVar 1@2) [0@0]),
        Send 0@0 (ChanInst (TVar 1@0) [0@0,0@0]),
        Buffer 0@0 (True,0,0)
    ]
)))

Estado 1:

(New (-1) (bind t03 (
    Par [
        Recv 0@0 (ChanInst (TVar 1@2) [0@0]),
        Send 0@0 (ChanInst (TVar 1@0) [0@0,0@0]),
        Buffer 0@0 (True,0,0)
    ]
)))
```

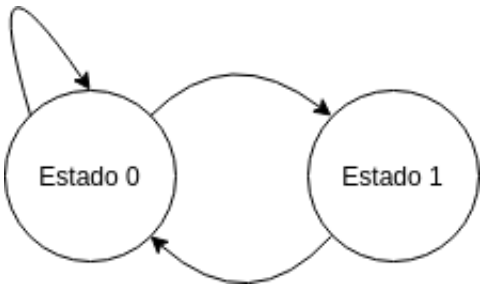


*Fig. 2.15:* Estados de la LTS generados en orden a partir del ejemplo de la figura 2.1

- invocar en paralelo la función `recv` que genera una elección interna (`IChoice`)
- ingresar al ciclo y escuchar sobre el canal creado (`Send`)

En este estado la computación sólo puede avanzar si se ejecuta alguna de las dos acciones "silenciosas"del condicional. Si se ejecuta la segunda de las acciones silenciosas (no se cumple la condición del `if`), se reingresa al ciclo y vuelve a aparecer el mismo condicional, por lo que se vuelve a llegar nuevamente al mismo estado 0. Si en cambio se ejecuta la primer acción silenciosa (se entra al condicional) se reemplaza la elección interna en la composición en paralelo por un tipo `Recv` sobre el mismo canal (0@0) por el que trata de enviar en el `Send` que ya estaba en la composición. Se llega así al estado 1, del cual sólo se puede volver al estado 0 mediante la sincronización de esas acciones de enviar y recibir.

#### 2.4.2.3.1 Dos funciones

La generación del LTS se realiza mediante dos tipos de ejecución: la que avanza todo lo posible sin generar un estado nuevo y la que agrega un estado nuevo al LTS.

La función que realiza el primer tipo de ejecución es `normalise`, la cual inicia canales y *buffers* y luego invoca a la función `unfoldTillGuard` que realiza el resto del trabajo. Ésta última no sólo crea canales y reemplaza las invocaciones a función tanto secuenciales como en paralelo por el tipo correspondiente, sino que también detiene la ejecución si así lo exigen las reglas de ejecución semántica acotada por una cota $k$ (para más información ver [8]). En este paso sólo se consumen las instrucciones `Par`, `ChanInst (TVar)`, `New`, `ChanAbst` y `Seq`. Si el tipo no es ninguno de estos (es una instrucción que genera barbas) la ejecución termina y se devuelve el tipo sin realizar ninguna modificación. Lo mismo sucede si el tipo es una invocación a una función con un determinado conjunto de canales que supera la cota establecida o si la función se está invocando por segunda vez en el marco de esa misma ejecución de `unfoldTillGuard`, lo cual significa que se está recorriendo un patrón que ya fue visto y no se debe avanzar más.

En términos generales `unfoldTillGuard` devuelve el tipo de una composición en paralelo con todos los tipos que pueden estar listos para sincronizar en un determinado estado, incluyendo los *buffers* de los canales nuevos creados.

La función que genera los estados del LTS es `succs`. Ésta utiliza la función `genStates` que toma una lista de estados (en este caso, el estado inicial como único elemento) y genera todos los estados nuevos que puede a partir de estos. Luego los asocia a las definciones de funciones dentro del *scope* del programa y los devuelve.

La función `genStates` toma una lista de estados ya recorridos y una lista de estados por recorrer. Cada vez que recorre un estado pasado por parámetro lo saca de la lista de estados por recorrer y lo agrega a la lista de estados ya recorridos. Al generar un nuevo estado, se invoca recursivamente agregando el estado generado a la lista de estados por recorrer (para ver si puede generar más estados a partir de éste) sólo si ese nuevo estado generado *no está* en la lista de estados ya recorridos (no genera estados repetidos).

Por último, `genStates` puede generar varios estados nuevos en simultaneo a partir de calcular las guardas del estado recorrido y ver todas las que pueden sincronizar. Cada guarda se representa mediante una tupla que tiene la acción de sincronización seguida por el tipo que continua la ejecución. Al realizar una sincronización se genera un estado nuevo reemplazando la instrucción que generó la guarda por el segundo elemento de la tupla.

### 2.4.3. Implementación del chequeo de Liveness

En **Gong**, el chequeo de **liveness** se realiza mediante la función `liveness` [22] que toma un parametro para *debug*, una cota $k$ y una lista de tipos `Eqn` que representa a la LTS generada mediante la ejecución simbólica del tipo del programa a chequear. La función chequea para cada uno de los estados del LTS y para cada tipo compuesto en paralelo dentro de cada estado si existe una correspondencia entre barbas (una acción de sincronización posible) con alguno de todos los otros estados y tipos del LTS.

Para obtener una lista con los tipos compuestos en paralelo en un determinado estado se invoca la función `extractType` que ignora las declaraciones de canales y devuelve la lista de tipos compuestos en paralelo si el tipo recorrido es una composición en paralelo o

una lista con el tipo recorrido como único elemento si no.

Luego llama a la función `checkStates` pasándole la lista generada. `checkStates` recorre cada uno de los tipos de la lista y por cada uno, genera un nuevo LTS a partir del resto de los tipos. El LTS generado representa a todos los estados alcanzables mediante todo tipo de reducciones y sincronizaciones sin utilizar el tipo actual. Luego, mediante la función `findMatch` se intenta sincronizar las barbas del tipo actual con las de alguno de los estados del LTS generado y ver si se valida el predicado de *liveness*, es decir, si existe alguna sincronización en algún punto futuro de la ejecución. Este chequeo se realiza para cada uno de los tipos de la composición en paralelo de cada uno de los estados del LTS original. Es importante que así sea para poder descubrir posibles *deadlocks* parciales que no obstaculizan el progreso del programa analizado en su totalidad pero tienen al menos un hilo de ejecución que no progresa. Es decir, no alcanza con chequear que desde el estado del LTS actual no existe una transición hacia ningún otro estado (lo que sólo probaría la existencia de un *deadlock* total), sino que hay que chequear que todos los hilos de ejecución paralelos realizarán una sincronización en algún momento de la ejecución futura.

La sincronización de las barbas entre dos tipos se analiza en la función `matchTypes`, la cual toma dos tipos, calcula las barbas para cada uno y luego devuelve el resultado de llamar a la función `synchronise` para el conjunto de barbas de cada uno.

El calculo de barbas se realiza en la función `barbs` y devuelve una lista con el mismo tipo como único elemento para el caso de un GoType de tipo `Send` o `Recv`. Para GoTypes de tipo `OChoice` o `Par` devuelve una lista con la llamada recursiva de la función aplicada a todos sus elementos. Para el caso de un tipo `New`, se llama recursivamente para el tipo siguiente. Para el caso de un tipo `Buffer`, devuelve una lista con un `Send` y un `Recv` en caso de que el tamaño del *buffer* sea mayor a 0 y menor a la capacidad del canal, o una lista con un `Send` solo o `Recv` sólo en caso de que la cantidad de elementos sea mayor a cero pero no menor a la capacidad del canal o viceversa respectivamente. En caso de que el canal esté vacío pero esté cerrado, devuelve una lista con un `Send`. Caso contrario, devuelve una lista vacía.

La sincronización genera un producto cartesiano entre las barbas de uno y otro de los dos tipos a sincronizar. Luego toma ese producto y lo filtra según aquellas tuplas compuestas de barbas que son compatibles entre sí. Dos barbas son compatibles si su composición en paralelo cumple el predicado $\downarrow_{[a]}$, es decir si una es un `Send` y la otra un `Recv`. Notar que la función `match` también contempla el caso de un `Close` y un `ClosedBuffer`, sin embargo, en este caso nunca se realiza esta comparación debido a que la función que genera las barbas para *liveness* no genera estas sentencias. La sincronización es verdadera si luego de filtrar la lista de los pares de barbas que *matchean*, el resultado no es una lista vacía.

### 2.4.4. Implementación del chequeo de Safety

El chequeo de **safety** se realiza mediante la función `safety` [23]. Ésta, al igual que `liveness`, toma una lista con todos los estados del LTS que representa la ejecución del tipo del programa y recorre cada instrucción compuesta en paralelo dentro de cada estado. Cuando encuentra una instrucción `Close`, calcula un nuevo LTS a partir de reemplazar

ese `Close` por un `Buffer` cerrado del mismo canal que el `Close` por un lado, y el tipo de la instrucción que continua la ejecución luego del `Close` por el otro. El procedimiento es similar al de la función `liveness` sólo que en lugar de generar un LTS nuevo sacando uno de los tipos de la composición en paralelo, lo genera reemplazando cada `Close` posible por el resultado de consumir ese `Close`.

A continuación compara las barbas de cada tipo `Close` consumido con las de cada estado del LTS generado posteriormente. Si se encuentra que alguna combinación de estas rompe la propiedad de la figura 2.8 toda la función devuelve *false* y se informa que el sistema no satisface **channel-safety**. Las barbas del `Close` siempre son una lista con un `Close` como único elemento, mientras que las barbas de cada estado del LTS generado para ese `Close`, son una lista con todos los `Send` y `Close` que pueda tener en paralelo. La comparación de las barbas se hace mediante la función `noclose` que calcula el producto cartesiano de las dos listas de barbas pasadas por parámetro y devuelve verdadero si ese producto filtrado por un `filter` que se queda sólo con los pares que hagan una combinación incorrecta, es vacío. La combinación incorrecta se calcula aplicando la función `badmatch` que devuelve verdadero si se le pasan dos tipos `Close` o un `Close` y un `Send` sobre el mismo canal.

### 2.4.5. Sincronización con *buffers*: extensión al modelo de la comunicación asincrónica

Durante la reducción semántica, la extensión al modelo de la comunicación asincrónica se da fundamentalmente incorporando las acciones de un *buffer* con capacidad al momento de realizar una sincronización para pasar de un estado de la LTS al siguiente. Con este objetivo, la función `getGuardsCont` [24], que devuelve una tupla con una instrucción candidata a sincronizar en primer lugar y el tipo que continuará siendo ejecutado luego de la sincronización en segundo, devuelve para el tipo `Buffer` un conjunto distinto de guardas según los atributos del tipo. Hay tres tipos de guardas que pueden devolverse para este tipo:

1. Una tupla con `ClosedBuffer` como primer elemento y un `Buffer` sobre el mismo canal que el original con la misma capacidad y tamaño pero cerrado como segundo (cerré el *buffer*).

2. Un `Send` sobre el mismo canal del *buffer* en primer lugar y el mismo *buffer* original pero restándole uno a la cantidad de elementos en segundo (saqué un elemento del *buffer* y lo mandé al canal).

3. Un `Recv` también sobre el mismo canal seguido del mismo *buffer* original pero sumándole uno a la cantidad de elementos (recibí de un canal y lo guardé en el *buffer*)

Las guardas serán devueltas según los siguientes valores del atributo `Buffer`: si el *buffer* tiene capacidad y cantidad de elementos igual a cero, se devuelve una lista con sólo la guarda 1. Si el *buffer* tiene cantidad de elementos mayor a cero y capacidad mayor a la cantidad de elementos, se devuelve una lista con las tres guardas posibles. Si el *buffer* tiene cantidad de elementos mayor a cero pero no menor que la capacidad del canal devuelve sólo las guardas 1 y 2 (`Send` y `ClosedBuffer`). Si en cambio, el canal tiene 0 elementos

pero una capacidad mayor a cero devuelve las guardas 1 y 3 (`Recv` y `ClosedBuffer`). Por último, si el *buffer* está cerrado, devuelve las guardas 1 y 2 (`Send` y `ClosedBuffer`).

A su vez, durante el chequeo de **liveness**, se agregan también las nuevas barbas correspondientes a los *buffers* con capacidad en la función `barbs`, que devuelve los predicados validados por cada tipo. Acá, debido a que sólo se está trabajando con las barbas relativas a la validación o refutación de la propiedad de **liveness**, se puede devolver para un *buffer* sólo combinaciones de operaciones de `Send` y `Recv` sobre su canal correspondiente. No interesan en este caso los predicados relativos a la acción de cerrar un canal. La función `barbs` devuelve para un *buffer* con tamaño mayor a cero y menor a la capacidad total, una lista con `Send` y `Recv`. Si el *buffer* tiene tamaño igual a cero pero capacidad mayor devuelve una lista con sólo `Recv`. Si por el contrario, tiene tamaño mayor a cero pero no menor a la capacidad del canal o si, por otro lado, el canal está cerrado, devuelve una lista sólo con `Send` indicando que puede enviar mensajes si otro tipo lo solicita.

# 3. TRAZAS

En el siguiente capítulo se detallan los cambios realizados en las herramientas **Dingo-Hunter** y **Gong** para devolver las trazas de las posibles ejecuciones que puedan romper la propiedad de *liveness* y/o *safety* en un programa. En la sección 3.1 se describe el trabajo realizado para incorporar el número de línea en los programas MiGo y en el extractor que los genera. En la sección 3.2 se detalla qué estructuras nuevas se agregan a la implementación del tipo sesión para poder construir la traza, cómo se inicializan esas estructuras con la información relativa al número de línea incorporado al código MiGo y qué cambios se realizan en la ejecución simbólica del tipo para ir acumulando la historia e información relativa a la ejecución. También se describe un problema surgido en relación a la $\alpha$-equivalencia de tipos iguales con distinta traza. En la sección 3.3 se aborda la cuestión de cómo mostrar una traza de ejecución fallida una vez que se identifica un problema. Por último, en la sección 3.4 se explica particularmente cómo se abordó la sincronización con canales con *buffers* de capacidad mayor a cero explicando decisiones tomadas respecto a qué información mostrar. El proyecto con el aporte realizado puede encontrarse en [25] y [26].

## 3.1. Cambios en el extractor de MiGo

El objetivo planteado sobre las modificaciones a realizar en el extractor de MiGo fue agregar al resultado la información sobre el número de línea en el código Go original. De esta manera, esta información puede ser incorporada luego al tipo generado con **Gong**. Se decidió incorporarlo al final de cada instrucción antes del caracter ';' que marca el final de cada línea y precedida a su vez por el caracter '@' para indicarle al *parser* que a continuación debe esperar un número de línea. En la figura 3.1 podemos ver el resultado de ejecutar la extracción sobre el programa de la figura 2.1 luego de las modificaciones aplicadas. Se destacan con color rojo los cambios respecto a la versión anterior que se muestra en la figura 2.10. Podemos ver que se reconoce la línea 4 donde se crea el nuevo canal como así también la línea 6 donde se invoca la función `recv` en paralelo. También dentro del ciclo de la función `main` se reconoce la línea 8 donde se envía sobre el canal creado y por último, se reconoce también la línea 15 donde se recibe de ese canal.

Las instrucciones con número de línea 0 son instrucciones que no se corresponden con ninguna instrucción del código original. Estas instrucciones se generaron al convertir el programa a su representación SSA el cual modela las estructuras de control de flujo `for` y `while` mediante el uso de saltos o saltos condicionales. Si bien en la representación SSA los saltos y las invocaciones de llamadas son distinguidas mediante tipos de instrucción distintos (`jump` y `call` respectivamente), en MiGo esta distinción no existe y por lo tanto ambos tipos se mapean al mismo tipo de instrucción `call`. Es necesario distinguir qué instrucciones generadas se corresponden con instrucciones presentes en el código tomado como entrada del programa y qué instrucciones no para no generar información sobre instrucciones inexistentes que sólo aportarían confusión al usuario de la herramienta. Se decidió por lo tanto, utilizar el número 0 como marca. De esta manera, Gong no va a agregar información sobre la invocación de esta función al armar las trazas de ejecución.

```
def main.main():
    let t0 = newchan main.main.t0_0_0, 0 @4;
    spawn main.recv(t0) @6;
    call main.main#1(t0) @0;
def main.main#1(t0):
    send t0 @8;
    call main.main#1(t0, t0) @0;
def main.recv(ch):
    call main.recv#1(ch) @0;
def main.recv#1(ch):
    if recv ch @15; call main.recv#1(ch) @0;
        else call main.recv#1(ch) @0; endif;
```

*Fig. 3.1:* Resultado de ejecutar **dingo-hunter** con las modificaciones realizadas sobre el código Go de la figura 2.1

### 3.1.1. Implementación

A continuación describimos los cambios principales para cada etapa de ejecución de la herramienta **Dingo-Hunter** descripta en 2.4.1 y esquematizada en la figura 2.9.

1. La primera etapa, el pasaje del código tomado como entrada a su representación SSA, se hace mediante la librería SSA de Go [21]. Esta librería ya incorpora el número de línea con lo cual no es necesario aplicar ninguna modificación.

2. La segunda etapa se da al pasar de la representación SSA a la representación abstracta de MiGo conformada por los *statements* de las instrucciones. Aqui se realizaron dos modificaciones:

   - La primera consiste en adaptar las diferentes estructuras que implementan la interfaz *Statement* correspondientes a cada instrucción para poder incorporar la información nueva. Con este propósito se agrega un campo LineNum de tipo `string` en cada una.
   - Se modifica la función `visitInstr` para obtener el número de linea y agregarlo a cada uno de los *statmentes*.

3. La tercer etapa se da al imprimir la representación del programa MiGo (conformada por los *statements* agrupados en `Functions`) en texto. En esta etapa, modificamos las funciones `String` de cada uno de los *statements* agregando el número de línea antecedido por el caracter '@' al final de la instrucción y antes del caracter ';' que indica el fin de la sentencia.

Con respecto al primer punto del item 2, se eligió el tipo *string* para el campo `LineNum` debido a que el valor que tiene al no estar inicializado es el *string* vacío. De esta manera, si existe algún error que trae como consecuencia que la estructura no se inicialice como corresponde, al generar el código MiGo en la última transición de la figura 2.9 obtendremos el caracter '@' seguido directamente por el caracter ';' que finaliza la sentencia.

```
type CallStatement struct {
        Name    string
        Params  []*Parameter
        LineNum string
}

func (s *CallStatement) String() string {
        return fmt.Sprintf("call␣%s(%s)@%s",
                s.SimpleName(), CallerParameterString(s.Params), s.LineNum)
}
```

*Fig. 3.2:* CallStatement después de las modificaciones realizadas

Esta situación se identifica como un error en la generación del código MiGo y se busca diferenciarla, por ejemplo, del caso de una invocación marcada con número de línea 0 que no representa un error. Esta diferenciación no sería posible si el tipo del campo LineNum hubiera sido, por ejemplo, de algún tipo numérico cuyo valor sin inicializar es el 0. En la figura 3.2 se puede ver cómo queda la definición del *statement* de la instrucción `call` presentada como ejemplo, con los agregados realizados resaltados en rojo.

Con respecto al segundo punto del item 2 (los cambios realizados en la función `visitInstr`) es necesario considerar que la representación SSA no guarda el número de línea directamente sino que guarda un valor que representa al mismo tiempo la referencia al archivo donde está la instrucción y su *offset* desde la línea 0 de ese archivo. Para recuperar el número de línea fue necesario, por lo tanto, obtener primero el conjunto de archivos (*File Set*) accediendo a la variable `FSet` de la metadata de la inferencia de tipos realizada al construir la representación SSA. Luego, pasándole a la función `Position` del tipo `FSet` el valor guardado en la instrucción que está siendo recorrida obtenemos un *string* con el número correspondiente al archivo y el número del *offset* separados por el caracter ':'. Ese *string* se manipula luego para obtener el número de línea final que será incorporado al *statement* de MiGo.

### 3.1.2. Instrucciones sin número de línea

La información sobre el número de línea no se agregó en todas las instrucciones de MiGo. Se consideró sólo aquellas instrucciones que pudieran aportar información relevante en la construcción de una historia de ejecución. En este sentido, se modificaron los *statements* correspondientes a las instrucciones `recv`, `send`, `tau`, `close`, `spawn`, `call` y `newchan`. Se consideró que los números de línea de las instrucciones `if` y `select` en cambio no aportan a construir la historia de una ejecución (a lo sumo sirven los números de línea de las instrucciones que puedan tener dentro) y por eso sus *statements* no se modificaron.

## 3.2. Cambios en las estructuras de Gong

A continuación se presentan los cambios realizados en la herramienta **Gong** para generar las historias de ejecución a partir de la ejecución simbólica de un tipo. Tomando como base la figura 2.12 pueden separarse los cambios en tres partes:

1. **Representación intermedia**: durante el pasaje de un código MiGo a una representación abstracta intermedia, se modifica por un lado, el *parser* del código para que lea la información agregada luego de los caracteres '@' y por el otro, la estructura de la representación intermedia en sí para que contengan este nuevo valor.

2. **GoType**: se modifica la estructura de los GoTypes agregando un campo que será utilizado para que cada uno pueda ir construyendo su historia de ejecución durante la generación del LTS. Ese campo nuevo se inicializa con los números de línea tomados de la representación intermedia durante la etapa de chequeo y construcción del tipo.

3. **LTS**: se modifican las funciones que construyen el LTS para armar las historias de ejecución en cada estado. En esta etapa se realizaron dos modificaciones principales:

   - modificar la función `unfoldTillGuard` para que tome un nuevo parámetro que acumule las invocaciones a funciones realizadas y las agrege a la historia de cada tipo al devolverlo
   - modificar las funciones que generan las guardas y las sincronizaciones entre ellas para que agreguen la información sobre la sincronización realiada en el tipo que continua la ejecución.

Los cambios realizados respecto al último paso representado en la figura (la verificación de las propiedades de *liveness* y *safety*) se desarrolla en las proximas secciones de este capítulo.

### 3.2.1. Expansión de la representación intermedia

En la figura 3.2.1 se observa el tipo de la representación intermedia con los agregados realizados para incorporar el número de línea en color rojo. Luego, al *parsear* el código MiGo tomado como input, se busca al final de cada sentencia el caracter '@' y luego se utiliza la librería `Text.ParserCombinators.Parsec` y sus instrucciones "many" y "digit" combinadas para identificar una secuencia de digitos (cualquier cosa que sea reconocido por la expresión regular [0-9]+). La secuencia de dígitos se convierte luego al tipo *Int* mediante la instrucción `read` y se incorpora al tipo de la representación intermedia.

### 3.2.2. Historia de ejecución en cada GoType

Para construir las historias de ejecucion previas a cada instrucción se incorpora a los constructores del tipo *GoType* un parámetro adicional de tipo *String*, como se muestra en color rojo en la figura 3.4. Notar que, a diferencia de la modificación realizada para las representaciones intermedias de la figura 3.2.1, el parámetro adicional es en este caso, una *string* y no un entero. Esto se debe a que interesa no sólo mostrar un número de línea sino toda la traza de la ejecución en un momento dado para lo cual un entero resulta insuficiente.

A la vez, no todos los constructores tienen el parámetro *String* adicional. Hay dos motivos por los que se decidió no agregar este parametro. El primero es que el constructor no se corresponde con una instrucción real del programa Go original por lo que no aportan información propia y a la vez no necesitan acumular tampoco información para pasarla a un posible tipo siguiente en la ejecución. Esto vale para los constructores `Null`, `Buffer`

```
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. 3.3:* Representación intermedia del código MiGo parseado antes de que se realize el chequeo de tipos

```
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)
```

*Fig. 3.4:* GoTypes definidos en Gong depsués de los cambios realizados

y `ClosedBuffer`. Una razón similar aplica al constructor `ChanAbst` que si bien se corresponde con la definición de una función en el programa Go original, no resulta relevante la información sobre en qué línea aparece la definición y a la vez, como con los constructores anteriores, éste nunca aparece antes de otra instrucción a la que pueda tener que pasarle información.

El segundo es que una instrucción real del programa Go original se representa mediante la composición de dos *GoTypes* distintos, por lo que se decidió agregar la información en solo uno de los constructores para evitar redundancia. Este es el caso de `ChanInst` que junto con `TVar` representan la invocación a una función: mientras que `ChanInst` representa la instanciación de los canales que la función toma como parámetros, `TVar` representa el nombre de la función invocada que debe estar presente entre las definiciones disponibles del programa ejecutado. Se decidió agregar el campo *string* sólo en el constructor `TVar`.

Al generar los GoTypes a partir de la representación intermedia, se dan tres casos en términos de cómo se inicializa el campo agregado:

- Para las instrucciones `IChoice`, `OChoice` y `New` no hay información relevante que agregar al constructor y el *GoType* se genera con el *string* vacío, el cual será utilizado para acumular información y pasarla a las instrucciones siguientes.
- Para las instrucciones `Send`, `Recv`, `Tau` y `Close` se añade la información del número de línea sin ningún tipo de información extra.
- Para las instrucciones `Par` y `Seq`, que siempre se construyen con una lista con el tipo de la invocación a una función como primer elemento (`TVar`) seguido del tipo que continua la ejecución, inicializamos el campo de la traza con el string vacío. Luego agregamos el número de línea al `TVar` antecedido por el texto *"CALL on Line"* o *"SPAWN on Line"* según corresponda. Agregar esta información en este lugar simplifica luego la construcción de la traza durante la ejecución simbólica ya que permite distinguir los tipos `TVar` según si fueron introducidos mediante el lanzamiento de un nuevo hilo de ejecución o no. La excepción es el caso donde el número de línea de la invocación esté marcado con el 0, en cuyo caso se deja el *string* de `TVar` vacío.

### 3.2.3. Ejecución Simbólica

La ejecución simbólica del tipo proporciona la información necesaria para terminar de armar las trazas. Registramos dos tipos de pasos de reducción durante la ejecución simbólica: la invocación a una función y la sincronización de acciones sobre un mismo canal. Esto se logra modificando las fases de generación de estados del LTS (las dos funciones que se desciben en la sección 2.4.2.3.1). En la primera, donde se ejecutan las invocaciones a funciones, se agrega un parametro que acumula información sobre éstas en las llamadas recursivas y la agrega al tipo final devuelto. En la segunda, se agrega al tipo que sigue a una sincronización, la información sobre ésta.

Por otro lado, la modificación de los GoTypes rompe la implementación de la $\alpha$-equivalencia ya que dos tipos iguales pero con distinta historia de ejecución dejan de ser considerados equivalentes. Esto provoca que se empiecen a generar LTS infinitos. Por este

motivo, se redefine la función de $\alpha$-equivalencia con el fin de excluir el campo agregado a los tipos de la comparación.

#### 3.2.3.1. Reducción sin generar un nuevo estado

En la función `unfoldTillGuard` agregamos un parámetro *String* llamado *trace*. Éste parámetro será utilizado para pasar información ya generada sobre la ejecución previa a los tipos subsiguientes. Las llamadas recursivas de la función siempre utilizan el valor del *trace* anterior. Si recorremos el tipo de una invocación a una función, al *trace* anterior se le agrega entonces la información sobre ésta en la llamada recursiva sobre el tipo asociado al nombre de la función invocada. Al hacerlo, no es necesario identificar si es una composición en paralelo o no ya que esta información ya fue agregada al generar el tipo.

Cuando ya no se puede reducir más sin realizar una sincronización y por lo tanto generar un nuevo estado del LTS entonces se agrega la información acumulada al nuevo tipo generado.

#### 3.2.3.2. Informacion a partir de la sincronización y generación de un nuevo estado

Cuando se realiza una acción de sincronización sobre tipos compuestos en paralelo y se genera por consiguiente un nuevo estado se debe agregar al tipo del nuevo estado generado la información sobre la sincronización hecha. Hay, también, dos momentos distintos de la sincronización que importa analizar. El primero tiene que ver con la comunicación asincrónica y será tratado en la sección 3.4 relativa a la extensión del modelo para este caso.

El segundo se da en el momento de hacer efectivamente la sincronización. La función `compatibleConts` toma dos listas de guardas compuestas por tuplas con la instrucción sobre la que se busca sincronizar y el tipo a ejecutarse luego de la sincronización. Si la sincronización es posible, una vez consumidos los tipos de la operación sincronizada, agregamos la información correspondiente mediante la función `getLineFromSynched` al tipo siguiente que formará parte del siguiente estado.

La función `getLineFromSynched` toma dos tipos. Para el primero agrega información sobre la operación a realizar distinguiendo entre `Send`, `Receive` y `Tau`. Luego, agrega la información acumulada en la traza del segundo tipo agregando un nivel de indentación mediante la función `replaceNewLines`. Como resultado, tenemos la información acerca de la instrucción perteneciente al hilo de ejecución que venía ejecutandose en un primer nivel de indentación y la información sobre la ejecución de la instrucción que sincronizó con ésta en un segundo nivel de indentación. Como estamos incluyendo el historial completo de la segunda instrucción, es posible que aparezcan más niveles de indentación siempre y cuando en esta instrucción se incluya información sobre sincronizaciones previas.

Se decidió no considerar en esta instancia la información sobre sincronizaciones realizadas entre un hilo de ejecución que cierra un canal y el *buffer* que recibe la petición. La razón es que no se corresponde con una sincronización real que se refleje en el código Go original sino con cómo **Gong** modela una acción interna del compilador de Go. Por lo tanto, resulta imposible indicar un número de línea o cualquier tipo de información que se

corresponda con el *input* de la herramienta y que por lo tanto pueda servirle al usuario.

#### 3.2.3.3. $\alpha$-equivalencia

Cuando la función `genStates` recorre los estados del LTS que representa la ejecución simbólica del tipo, chequea si el estado que está recorriendo (representado por un GoType) pertenece a la lista de estados ya vistos para evitar recorrerlo nuevamente y por lo tanto, generar potencialmente un LTS infinito con estados repetidos. Este chequeo lo realiza mediante la función `inList` [27], la cual toma un *GoType* y una lista de *GoTypes* y chequea el tipo contra cada elemento de la lista utilizando la función `aeq`. La función `aeq` (por *Alpha EQuivalence*) pertenece a la clase `Alpha` [28] del paquete de Haskell *Unbound-LocallyNameless* [29] y permite comparar tipos que puedan llegar a contener nombres. Los GoTypes construidos con `Send`, `Recv`, `New`, `Close`, `ChanInst`, `ChanAbst`, `Buffer` y `ClosedBuffer` toman un parámetro de tipo ChName donde ChName está definido como Name Channel usando el constructor Name del paquete *Unbound-LocallyNameless*. Los GoTypes construidos con `TVar`, además, toman un parámetro de tipo EqnName donde EqnName está definido como Name GoType, usando el mismo constructor. Por eso debe usarse esta función para comparar los nombres de los canales o de las funciones referenciados por los tipos.

La función `aeq` compara para dos constructores iguales de un mismo tipo, todas sus variables de tipo y en particular, permite comparar instancias de tipo `Name`. Para los GoTypes de la figura 2.14, dos operaciones de Send sobre un mismo canal, seguidas de un mismo GoType, por ejemplo, son $\alpha$-equivalentes. Sin embargo, para los GoTypes de la figura 3.4, dos operaciones de `Send` sobre el mismo canal y seguidas de un mismo GoType pero con distinta traza de ejecución no lo son. Dos `Send` ejecutados dentro del mismo patron de comunicación, en el mismo número de línea, seguidos de un mismo comportamiento pero con distinta traza (por ejemplo porque uno tiene una llamada recursiva más en su historial) son considerados distintos por la función `aeq`. Como resultado de esto, la modificación realizada en la figura 3.4 genera un LTS infinito y la ejecución nunca termina.

Para evitar este problema, es necesario redefinir la equivalencia de GoTypes desestimando las posibles diferencias que pudiesen existir en las trazas de cada tipo. Con este proposito, se intrudojo la función eqGT [30], la cual toma dos GoTypes y compara los valores de todos sus campos menos el correspondiente a la historia de ejecución. La función utiliza el comparador `aeq` para la mayoría de las comparaciones de los campos con la excepción de aquellos que son de tipo GoType, para los cuales se llama recursivamente.

## 3.3. Liveness y Safety

Las trazas guardan en cada instrucción, su historial de ejecución: todo lo que sucedió antes de llegar hasta donde está. Pero su propósito es poder brindar información sobre un tipo de historia de ejecución particular: las que no satisfacen las propiedades de *liveness* y *safety*. Por lo tanto, para cumplir su propósito, se debe poder identificar dónde falla la ejecución.

```
1. package main
2.
3. func main() {
4.         ch := make(chan int)
5.         go send(ch)
6.         <-ch
7.         <-ch
8. }
9.
10. func send(ch chan int) {
11.         ch <- 1
12. }
```

*Fig. 3.5:* Ejemplo de programa con *deadlock* en la línea 7

En la figura 3.5 se puede observar un programa que lanza una *goroutina* que envía sobre un canal y luego de lanzarla recive un mensaje sobre el mismo canal dos veces, generando un *deadlock* en la línea 7. En la figura 3.6 se ve el resultado de correr sobre este programa, la herramienta **Gong** con las modificaciones realizadas. La herramienta identifica la operación de recibir que no puede sincronizar en la línea correspondiente y luego muestra la historia de ejecución previa (la otra operación de recibir que sincronizó con una operación de enviar en la línea 11, que a su vez fue generada por una invocación en paralelo en la línea 5).

En la figura 3.7 se puede observar un programa que lanza dos *goroutinas* que envían sobre un canal, luego escucha una vez sobre éste y luego lo cierra, haciendo que potencialmente la segunda rutina pueda llegar a enviar sobre el canal cerrado generando un *panic error*. En la figura 3.8 se puede observar el resultado de correr sobre este programa, la herramienta **Gong** con las modificaciones realizadas. **Gong** identifica el cierre del canal en la línea 8 colisionando con la operación de enviar sobre el canal en la línea 12 agregando luego la historia de ejecución previa de cada una de las dos operaciones. Vale la pena mencionar que Gong también reconoce este programa como no *live* debido a que el segundo *send* no sincroniza.

El fallo de una ejecución está definido por las barbas del tipo de la figura 2.7. Para identificar las ejecuciones fallidas y mostrar su información fue necesario modificar las funciones que validan las barbas para que dejen de devolver un *booleano* indicando que se cumple o no con una propiedad y devuelvan una lista de instrucciones que hacen que la propiedad deje de cumplirse (o la lista vacía en caso contrario). En las secciones siguientes se detalla cómo se implementó esto para las verificaciones de *liveness* y *safety* respectivamente.

### 3.3.1. Liveness

La función que identifica para cada estado que sea *live* es `checkStates`, que toma una lista de los tipos compuestos en paralelo en ese estado y devuelve un *booleano* indicando

```
There is a Recv operation without synch on line 7
RECV on line 6
        SEND on line 11
        SPAWN on Line 5
```

*Fig. 3.6:* Mensaje mostrado por la herramienta *Gong* al analizar el programa de la figura 3.5

```
1. package main
2.
3. func main() {
4.         ch := make(chan int)
5.         go send(ch)
6.         go send(ch)
7.         <-ch
8.         close(ch)
9. }
10.
11. func send(ch chan int) {
12.         ch <- 1
13. }
```

*Fig. 3.7:* Ejemplo de programa que envía en la línea 12 sobre canal cerrado en la línea 8

```
Safety: Term not safe:
There is a close operation on line 8
RECV on line 7
        SEND on line 12
        SPAWN on Line 5

Colliding with:
A send operation on line 12
SPAWN on Line 6
```

*Fig. 3.8:* Mensaje mostrado por la herramienta **Gong** al analizar el programa de la figura 3.7

si todos los tipos de la lista van a eventualmente sincronizar con algún otro tipo en una posible ejecución futura. El *booleano* devuelto se calcula como la conjunción de chequear la propiedad para cada uno de los elementos de la lista. Para los tipos recorridos, en lugar de devolver verdadero en caso de que sincronice, vamos a devolver el tipo recorrido en caso de que **no** sincronice, concatenadolo al resultado de la llamada recursiva, dado que podría existir más de un tipo dentro de la composición que no sea *live*. De esta manera, en vez de obtener un *booleano*, vamos a obtener una lista de GoTypes que *no* sincronizan. Por lo tanto, es necesario adaptar la función `liveness` que llama a `checkStates` para que espere el nuevo valor pero evitando que modifique su significado debido a este cambio. La forma de hacer esto es tomar el valor verdadero si la lista de GoTypes que no pudieron sincronizar con otros estados es vacía y tomar el valor falso si no. Esto se hace mediante la consulta `null` sobre la lista resultado la cual reemplaza al *booleano* anterior.

Una vez realizada esta modificación, al finalizar la ejecución de la función `liveness`, tendremos ya calculada la lista de los posibles GoTypes que representan la última instrucción de una posible historia de ejecución fallida. Como cada GoType tiene su historial de ejecución, lo único que necesitamos es mostrarlo. Sin embargo, estos GoTypes sólo pueden ser de tipo `Send` y `Recv`, los cuales, hasta ahora sólo completaron su información relativa a su historia de ejecución cuando sincronizaron. Como la instrucción encontrada no sincronizó, su información relativa a la traza está incompleta. Se debe entonces, completarla en este punto final de la ejecución indicando que ésta es la acción que falló, distinguiéndola de los `Send` y `Recv` anteriores.

Con este propósito se escribió la función `analyze` que intercala un separador entre el resultado de aplicar `notSynch` a cada elemento de la lista devuelta por `checkStates`. La función `notSynch`, por su parte, devuelve un texto indicando si la función donde falla la sincronización es un `Send` o un `Recv` y agregando luego, la traza de la instrucción.

### 3.3.2. Safety

La información que resulta útil para identificar posibles problemas relativos a la propiedad de *safety* es la combinación incorrecta de instrucciones donde primero se ejecuta un `Close` y luego se ejecuta o bien otro `Close` o bien un `Send` sobre el mismo canal que ya había sido cerrado. Se puede obtener esta información de la combinación incorrecta de barbas encontrada por la función `noclose`. La función `noclose` devuelve verdadero si el resultado de filtrar las combinaciones incorrectas de barbas es vacío. Sin embargo, es posible devolver en este punto de la ejecución la lista con esas combinaciones, sea vacía o no, en lugar de devolver simplemente un valor *booleano*. Luego será necesario cambiar el resto de las funciones que invocan a `noclose` (como `checkPair`, `checkList` y `checkAllSuccs`) para que devuelvan la suma de pares incorrectos de barbas de cada tipo ejecutando en paralelo en cada estado de la LTS pasada por parámetro.

Ahora en la función `safety`, en lugar de un *booleano*, se tiene una lista de pares con operaciones que rompen la propiedad **channel-safety**. Debemos por lo tanto modificar la función para que chequee si esa lista está vacía. Si no lo está, podemos utilizarla para devolver información que sirva para identificar las líneas de código que producen la ejecución erronea.

La primera dificultad que se presenta al respecto es que las barbas pueden aparecer repetidas. Como cada `Close` se compara con todos los estados del LTS generado a partir de él, es posible que la misma instrucción erronea ((Close) o `Send` sobre el mismo canal) aparezca en la composición en paralelo de más de un estado del LTS. Si existe un `Close` seguido de una operación erronea y en paralelo hay otros tipos que pueden sincronizar siguiendo otro hilo de ejecución, el LTS generado tendrá al menos dos estados que rompen la propiedad **channel-safety**: aquel donde se ejecuta primero el otro hilo de ejecución y luego la instrucción incorrecta y aquel donde se intenta ejecutar la instrucción incorrecta en primer lugar. Esto generará pares de barbas repetidos con exactamente las mismas instrucciones. Por eso se aplica un filtro utilizando dos `foldr` anidados para eliminar pares equivalentes en la lista de tuplas devuelta por la función `checkAllSuccs`. Luego se agrega la función `findCollidingOperations` para imprimir un texto identificando cada una de las operaciones incompatibles según su tipo y mostrando la traza de ejecución de cada una.

La segunda dificultad es que para que todas las operaciones incompatibles encontradas tengan su traza completa, es necesario pasar la información del `Close` consumido, a las operaciones que aparezcan después de éste. Caso contrario, cualquier operación incompatible generada en un orden de ejecución posterior al `Close` tendrá su traza incompleta. La operación que consume el close es `getContinuation`, que lo reemplaza por la composición en paralelo el tipo que sigue al close con un *buffer* cerrado. Para el tipo que sige al close, llamamos a la función `addToLine`, agregando en el tope de la traza la leyenda "Close operation on line " seguida de la línea del `Close` consumido.

## 3.4. Información sobre acciones internas: una historia de ejecución para la comunicación asincrónica

En la figura 3.9 observamos un programa que crea un canal con capacidad 2 y luego primero envía tres veces y luego recibe tres veces sobre él, generando un *deadlock* la tercera vez que envía. Al evaluar qué debería devolver la versión modificada de **Gong** en este caso nos encontramos con el problema de que no existen rutinas compuestas en paralelo sobre las que se hayan realizado sincronizaciones. Las sincronizaciones se realizaron entre la función principal y el canal de un *buffer* asincrónico. **Gong** modela este *buffer* como si fuera un hilo de ejecución en paralelo, pero esto no se corresponde con el programa original donde sólo existe un único hilo de ejecución. ¿Cómo debemos incorporar entonces la información sobre esta sincronización?

La primer idea surgida al respecto fue simplemente indicar que la sincronización se dio contra un *buffer*. Luego se decidió incorporar también información sobre el estado del *buffer* antes y después de realizar la acción de sincronización para ayudar a entender por qué y en qué momento no fue posible seguir ejecutando.

Para implementar esto, se modificó la función `getGuardsCont` para el caso de la generación de barbas del tipo `Buffer`. Los tipos que modelan las barbas que valida el *buffer* tienen como valor en el campo de su historia de ejecución el estado en el que quedaría el *buffer* si se sincronizara utilizando esa barba. Por ejemplo, si se sincroniza con la barba que estipula que el *buffer* de un canal es capaz de hacer un `Send` (porque tiene elementos que puede enviar si le son requeridos), la información de la traza de ese `Send` indicará que

```
1. package main
2.
3. func main() {
4.          ch := make(chan int, 2)
5.          ch <- 1
6.          ch <- 1
7.          ch <- 1
8.          <-ch
9.          <-ch
10.         <-ch
}
```

*Fig. 3.9:* Programa que envía tres veces sobre un canal con capacidad 2 y genera un *deadlock*

```
Liveness: Term not live:
There is a Send operation without synch on line 7
SEND on line 6
        RECV on line BUFFER: {Capacity: 2 - Size: 1 -> 2}
SEND on line 5
        RECV on line BUFFER: {Capacity: 2 - Size: 0 -> 1}
```

*Fig. 3.10:* Resultado de ejecutar **Gong** sobre el programa de la figura 3.9

la capacidad del *buffer* se disminuyó en uno. El caso del `Recv` es análogo pero indicando que la capacidad del *buffer* aumenta.

En la figura 3.10 está el resultado de correr **Gong** sobre este programa. La historia de ejecución devuelta identifica el *deadlock* en la línea 7 (la tercer operación de envío) y muestra la historia de sincronizaciones previas sobre el *buffer* donde se puede ver cómo aumenta su tamaño hasta llegar a su capacidad.

# 4. RESULTADOS

A continuación se analizan los resultados de los cambios introducidos a las herramientas **Gong** y **Dingo-Hunter** comentando decisiones tomadas y mostrando con ejemplos los objetivos alcanzados.

## 4.1. Historia de ejecución para ForSelect

En la figura 4.1 podemos observar la implemetación de la función `forselect`, que crea tres canales (`done`, `a` y `b`) y luego lanza dos ejecuciones paralelas pasandoselos como parámetros. Cada ejecución entra en un ciclo infinito donde realiza una elección externa (`select`) entre una operación de enviar y otra de recibir. Mientras que una ejecución envía del canal `a` y recibe de `b`, la otra envía de `b` y recibe de `a`. Si la sincronización se realiza sobre el canal `b`, cada *goroutina* vuelve a entrar en el ciclo y por lo tanto en el `select`. Si por el contrario, la sincronización se realiza en el canal `a`, cada *goroutina* envía un mensaje por el canal `done` y termina. Una de las dos envia el mensaje una vez, mientras que la otra envía dos veces. La rutina principal, en cambio, recibe sólo dos veces del canal, con lo cual la *goroutina* que envía dos veces sobre el mensaje `done` tiene un *deadlock* parcial (el resto de las rutinas finaliza).

En la figura 4.2 podemos observar el resultado de ejecutar **Gong** sobre el código MiGo extraído del programa de la figura 4.1. **Gong** detecta una operación de `Send` sin sincronizar en la línea 28 (el segundo mensaje enviado sobre el canal `done`). Previamente, se ejecutó un `Send` en la línea 27 que sincronizó con un `Recv` en la línea 42 (el primer mensaje enviado sobre el canal `done`), un `Send` en la línea 25 que sincronizó con un `Recv` en la línea 10 (los dos casos de accion dentro del `select` de cada *goroutina*) y una invocación en paralelo en la línea 39 (la invocación de la *goroutina* que da inicio al `select` infinito). A su vez, sobre el `Recv` de la línea 42, se puede decir también que previo a ejecutarse este, se ejecutó un `Recv` en la línea 41 que sincronizó con un `Send` en la línea 12 (el mensaje enviado sobre `done` por la otra rutina de `select` infinito) y que antes de este `Send` en la línea 12 se ejecutó un `Recv` en la línea 10 que sincronizó con un `Send` en la línea 25. Observar que en este punto se repite la misma historia que se puede ver en la dimensión vertical abajo del `Send` de la línea 27 que lanza esta nueva dimensión horizontal. Esto se debe a que el mismo hilo de ejecución que sincronizó luego en esta línea, lo hizo también antes con la misma *goroutina* que no valida el predicado de **liveness**.

## 4.2. Estructura de una traza

Las trazas acumulan la historia de todas las posibles distintas ejecuciones que puede tener un programa dentro de los límites de su *fencing*. Si para alguna de esas historias encuentra que alguna ejecución no es *live* o no es *safe*, se puede recuperar la historia de la ejecución partiendo del estado que viola la propiedad. Este es el último paso: identificar el punto de falla, completar su información y luego mostrar la historia que trae acumulada.

```
1. // Command nodet-for-select is a for-select pattern between
2. // two compatible recursive select.
3. package main
4.
5. import "fmt"
6.
7. func sel1(ch1, ch2 chan int, done chan struct{}) {
8.         for {
9.                 select {
10.                case <-ch1:
11.                        fmt.Println("sel1: recv")
12.                        done <- struct{}{}
13.                        return
14.                case ch2 <- 1:
15.                        fmt.Println("sel1: send")
16.                }
17.        }
18. }
19.
20. func sel2(ch1, ch2 chan int, done chan struct{}) {
21.        for {
22.                select {
23.                case <-ch2:
24.                        fmt.Println("sel2: recv")
25.                case ch1 <- 2:
26.                        fmt.Println("sel2: send")
27.                        done <- struct{}{}
28.                        done <- struct{}{}
29.                        return
30.                }
31.        }
32. }
33.
34. func forSelect() {
35.        done := make(chan struct{})
36.        a := make(chan int)
37.        b := make(chan int)
38.        go sel1(a, b, done)
39.        go sel2(a, b, done)
40.
41.        <-done
42.        <-done
43. }
```

*Fig. 4.1:* ForSelect con Deadlock en la línea 28

```
Liveness: Gong: Term not live:
There is a Send operation without synch on line 28
SEND on line 27
        RECV on line 42
        RECV on line 41
                SEND on line 12
                RECV on line 10
                        SEND on line 25
                        SPAWN on Line 39
                SPAWN on Line 38
SEND on line 25
        RECV on line 10
        SPAWN on Line 38
SPAWN on Line 39
```

*Fig. 4.2:* Resultado de ejecutar **Gong** para el `forselect` de la figura 4.1

La estructura de la historia de ejecución tiene dos niveles: un nivel vertical que representa un único hilo de ejecución (invocaciones realizadas o instrucciones ejecutadas previamente) y un nivel horizontal marcado mediante la indentación de la información, que representa un hilo de ejecución en paralelo cuya historia se relaciona a la historia del hilo principal a través de una determinada acción de sincronización. Puede haber varios niveles horizontales indicando varias sincronizaciones en distintos hilos de ejecución tal como se puede ver en el ejemplo de la figura 4.1.

El nivel vertical se construye principalmente en la función `unfoldTillGuard` al realizar la ejecución semántica sin sincronizaciones (sin generar un estado nuevo del LTS). La información en este nivel se lee como una pila donde el tope de la columna representa la información de la última instrucción ejecutada y el final representa el comienzo de la ejecución. Los elementos de la pila son todos aquellos que tengan un mismo nivel de indentación. En la historia de ejecución de la figura 4.2, por ejemplo, una pila está dada por la instrucción `Send` que genera el *deadlock* en la línea 27, la instrucción `Send` ejecutada antes en la línea 25 y la invocación en paralelo de una función (`sel2`) en la línea 25.

El nivel horizontal se genera al realizar una sincronización que genera un estado nuevo del LTS. Esta información se incorpora agregando un caracter de indentación posterior a todos los saltos de línea presentes en la historia de ejecución de la instrucción con la que se sincroniza. Si alguna línea de esa historia ya tenía un caracter de indentación luego de un salto de línea, se agrega uno nuevo generando subsiguientes niveles horizontales en la historia de ejecución. En la historia de la figura 4.2 se pueden observar hasta cuatro niveles horizontales demarcados por distintos grados de indentación que reflejan sincronizaciones realizadas en los distintos hilos de ejecución.

Al agregar niveles horizontales, es posible que una historia se repita en distintos niveles de una misma traza. Es lo que sucede también en el ejemplo de la figura 4.2 donde en un cuarto nivel de indentación y mostrando la sincronización de un `Recv` en la línea 10 con un

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.         ch := make(chan int, 1)
7.
8.         ch <- 1
9.         _ = <-ch
10.
11         close(ch)
12
13         _ = <-ch
14         ch <- 0
15         fmt.Println("This-is-not-safe")
16. }
```

*Fig. 4.3:* Función que recibe y envía sobre un canal cerrado con capacidad mayor a cero

`Send` en la línea 25 se repite una historia que aparece en el nivel de indentación principal de manera inversa (una operación de `Send` en la línea 25 que sincroniza con un `Recv` en la línea 10). Esto sucede porque dos hilos de ejecución distintos pueden sincronizar entre sí más de una vez en distintos momentos de su ejecución.

### 4.2.1. Falla de *safety* sobre canal asincrónico

En la figura 4.3 tenemos una función que crea un canal asincrónico de tamaño 1, y luego realiza las siguientes operaciones de manera secuencial: envía sobre el canal creado, recibe del mismo canal, lo cierra y luego recibe primero del canal cerrado y envía por último un cero por el mismo. En la figura 4.4 podemos encontrar el resultado de ejecutar **Gong** sobre el código MiGo extraido a partir del código de la figura 4.3. Allí podemos ver que se identifica a la operación de `close` de la línea 10 como colisionando con el envío de un mensaje por el canal cerrado en la línea 14. Al ver el resultado de la traza de la historia anterior, se muestra cómo se actualiza la información del *buffer* de capacidad 1 cuando se envía y se recibe sobre él y lo mismo sucede cuando se recibe del canal cerrado en la línea 13.

## 4.3. Correcciones en la implementación

Durante el analisis de las herramientas se descubrieron implementaciones que no funcionaban como se suponía. En la siguiente sección comentamos brevemente algunas de las fallas encontradas y qué solución se propuso

### 4.3.1. No se consideraban las operaciones $\tau$ dentro de un `select`

Cuando en un estado del LTS existía un `select` que tenía como una de sus guardas una acción de sincronización de tipo $\tau$ entre otras acciones de distinto tipo, **Gong** no era

```
There is a close operation on line 11
RECV on line 9
        SEND on line BUFFER: {Capacity: 1 - Size: 1 -> 0}
SEND on line 8
        RECV on line BUFFER: {Capacity: 1 - Size: 0 -> 1}
Colliding with:
A send operation on line 14
RECV on line 13
        SEND on line BUFFER: {Status: Closed}
CLOSE operation on line 11
RECV on line 9
        SEND on line BUFFER: {Capacity: 1 - Size: 1 -> 0}
SEND on line 8
        RECV on line BUFFER: {Capacity: 1 - Size: 0 -> 1}.
```

*Fig. 4.4:* Resultado de ejecutar **Gong** sobre el código MiGo extraído del programa de la figura 4.3

capaz de reconocer ese estado como *live*. Esto se debe a que en la función `barbs` que genera las barbas de cada instrucción no se generan barbas para el tipo $\tau$. Este comportamiento es correcto ya que el tipo $\tau$ no requiere barbas para sincronizar sino que se consume automáticamente. Por eso, para el tipo `Tau` la función `barbs` devuelve una lista de barbas vacía, lo que es interpretado como una instrucción que se puede consumir por sí sola sin validaciones extra.

Sin embargo, para el tipo `OChoice` que representa las instrucciones `select`, se devuelve una lista con las barbas de todas las acciones existentes en sus guardas. Si hay un `Tau`, por ejemplo, junto con un `Send`, se devuelve una lista con las barbas del `Send` solas ya que el `Tau` no genera barbas y **Gong** asume que la elección externa requiere sincronizar con alguna de las barbas del `Send` ignorando el `Tau`.

Este *bug* fue reportado a los implementadores los cuales modificaron la función `findMatch`, que llamaba luego a la función `barbs` creando un *match* especial para el tipo `OChoice` para el cual si alguna de las acciones de sus guardas es un `Tau`, la función devuelve inmediatamente verdadero, indicando que la instrucción es *live*.

### 4.3.2. Ejecución semántica para recibir sobre un canal cerrado

Durante la ejecución semántica del tipo, la función `getGuardsCont` devuelve las guardas de sincronización para cada tipo pasado por parámetro. Si el tipo era un `Buffer` cerrado, sin embargo, no se contemplaba correctamente la posibilidad de sincronizar con una operación de recibir, enviando el valor cero del tipo del canal. Por lo tanto, en los LTS generados nunca se realizaba la sincronización para este tipo de casos y no se creaban estados nuevos que pudieran surgir a partir de esta operación. Al chequear **liveness**, como en el último estado alcanzado existe un *buffer* cerrado y existe una operación de *receive* aplicada sobre ese *buffer*, las barbas estipulan que ese estado es *live*, pero si luego de realizar esa operación de recibir existe alguna operación que viole *channel-safety* o *liveness*, nunca se generará el estado de la LTS correspondiente y **Gong** nunca encuentra la falla.

Resolvemos este problema dividiendo en `getGuardCont` los dos casos en los que la cantidad de elementos del *buffer* es cero (si no lo es, siempre puede enviar un mensaje a quien lo solicite, esté cerrado o no) y si el *buffer* está cerrado, entonces va a producir una guarda de `Send` por más que no tenga ningún elemento.

## 4.4. Barbas para recibir sobre un canal cerrado

Se descubrió que **Gong** no contempla, al generar las barbas para chequear **liveness**, el caso de un `Buffer` sin elementos pero con capacidad mayor a cero que además esté cerrado. Sin importar si el *buffer* estaba cerrado o no, cuando el *buffer* no tiene elementos y tiene capacidad mayor a cero, se devuelve siempre la misma barba indicando que el *buffer* sólo puede simular una acción de recibir. Sin embargo, un buffer cerrado sin elementos puede enviar el valor *default* del tipo del canal si se ejecuta una acción de recibir sobre el canal al que está asociado. Para cubrir este caso, se dividió la guarda de la función `barbs` y en caso de que el tipo pasado por parámetro se corresponda con un `Buffer` cerrado, se devuelve también la barba correspondiente a enviar un mensaje ($\downarrow_{a^*}$)

# 5. CONCLUSIONES Y TRABAJO FUTURO

A continuación discutimos las principales conclusiones del trabajo realizado y delineamos aspectos que dejamos abiertos para trabajos futuros.

## 5.1. Conclusiones

La implementación de las historias de ejecución reutiliza la estructura de **Dingo-Hunter** y **Gong** por lo que no agrega una complejidad extra y permite mejorar su usabilidad identificando las instrucciones que pueden producir *deadlocks* o errores de comunicación. De esta manera, se facilita sustancialmente la corrección del código original.

Dilley y Lange [7] publican un estudio donde analizan el uso de primitivas de comunicación a través del intercambio de mensajes en Go con el objetivo de intentar determinar el impacto que las herramientas de análisis estático del código pueden tener sobre proyectos del mundo real. Los resultados indican que un 76 % de los proyectos analizados utilizan canales de comunicación con patrones de comunicación sencillos donde en la mayoría de los casos se utilizan canales sincrónicos. Esto demuestra el potencial que tienen las técnicas de verificación estática de programas como la desarrollada en [8] y en consecuencia, la importancia de las mejoras de usabilidad sobre estas herramientas que apunten a la masificación de su uso. Como prueba de concepto, **Dingo-Hunter** y **Gong** (y las modificaciones realizadas en este trabajo) aportan fundamentos al argumento de que la comunicación a través del intercambio de mensajes es una ventaja del modelo de Go debido a que abre la puerta al desarrollo de técnicas de verificación automática ([7]).

No todos los programas que utilizan canales, sin embargo, pueden ser verificados estáticamente. Por un lado, los programas que no son *fenced* no pueden ser analizados por el sistema de tipos propuesto en [8]. Por el otro, la satisfacción de una propiedad por un tipo no garantiza siempre que el programa representado por ese tipo también la satisfaga. En particular esto sucede cuando hay un condicional para el cual nunca se ejecuta uno de sus bloques. Supongamos, por ejemplo, que en el programa de la figura 2.1, en el condicional de la línea 14, en lugar de verificar que la variable `dummy` sea igual a 1, se verifica que sea igual a algún otro valor. Entonces nunca se ejecutará la acción de recibir sobre el canal pasado como parámetro y el programa tendrá un *deadlock*. Sin embargo, éste no sería detectado por **Gong** dado que, ignorando la condición, asumiría que ambos bloques del condicional se ejecutarán en algún momento de la ejecución.

En [7] también se identifica una gran cantidad de programas que utilizan una cantidad fija de canales de comunicación y para los cuales, sin embargo, no puede determinarse que utilicen también una cantidad finita de *gorutinas*. Para esta clase de programas, no existe aún una herramienta que verifique de manera estática la ausencia de errores de comunicación. Además, ciertos tipos de programas no pueden verificarse tampoco debido a problemas en la implementación de las herramientas que desarrolamos a continuación en las siguientes secciones.

```
package main
import "fmt"

func main() {
  for k := 0; false; k++ {
    fmt.Println(k)
  }

  for j:= 0;j < 3;j++ {
    fmt.Printf("%d␣", j)
    x := make(chan int)
    <-x
  }
}
```

```
def main.main#6():
  let t16 =
    newchan main.main.t16_0_0, 0 @11;
  recv t16 @12;
  call main.main#6(t16) @0;
  let t16 =
    newchan main.main.t16_0_1, 0 @11;
  recv t16 @12;
  call main.main#6(t16, t16) @0;
  let t16 =
    newchan main.main.t16_0_2, 0 @11;
  recv t16 @12;
  call main.main#6(t16, t16, t16) @0;
```

.

*Fig. 5.1:* MiGo representa los `loops` para los cuales puede calcular la cantidad de iteraciones mediante un `loop unrolling`

En la medida en que el desarrollo de los sistemas de tipos sesión abarque cada vez más tipos distintos de programas Go, el impacto que tengan en el uso del lenguaje y en los estilos de programar será mayor. El presente trabajo se inscribe en esta perspectiva buscando facilitar la programación en Go y la comunicación a través de intercambio de mensajes.

## 5.2. Correcciones en el extractor de MiGo

Existen algunos casos para los cuales el extractor del código MiGo en **Dingo-Hunter** devuelve una representación erronea del programa. En particular se detectaron un conjunto de problemas para representar de manera adecuada las estructuras de control para los siguientes casos:

- Cuando el ciclo tiene una cota definida (i $<$ 3 por ejemplo). En este caso, trata de hacer *loop-unrolling* pero agrega llamadas recursivas innecesarias sumando en cada *loop* un parámetro extra y haciendo que la llamada quede invalidada por tener una cantidad excesiva de parámetros (ver 5.1)

- Cuando hay *loops* anidados y ambos tienen cotas definidas la implementación realiza un *loop-unrolling* del ciclo interno e ignora la existencia del ciclo externo (ver 5.2)

Existen muchos programas que pueden caer en estos casos, lo que agrega una limitación extra al uso de la herramienta más allá de la limitación teórica planteada al rededor de la propiedad de *fencing*. Un posible trabajo futuro es, entonces, estudiar el pasaje de la representación SSA a la representación de MiGo intentando solucionar este problema.

```
package main
import "fmt"

func main() {
  for k := 0; false; k++ {
    fmt.Println(k)
  }

  for j:= 0;j < aux();j++ {
    fmt.Printf("%d", j)
    x := make(chan int)
    <-x
  }
}

func aux() int {
  return 3
}
```

```
def main.main#6():
 if let t16 = newchan main.main.t16_0_0, 0
  recv t16 @12;
  call main.main#6(t16) @0;
 else endif;

.
```

*Fig. 5.2:* Cuando MiGo no puede calcular directamente la cantidad de iteraciones de un `loop`, lo representa mediante un `jump` condicional

## 5.3. Problemas sobre canales

Los *deadlocks* parciales y/o totales y las operaciones incorrectas de semántica no son los únicos problemas que pueden ocurrir respecto al comportamiento comunicacional de un programa que utiliza canales para comunicarse. Existen un conjunto de problemas asociados al *leak* de recursos que pueden ser todavía más comunes incluso que los problemas de *liveness* y *safety* tratados en este trabajo. A continuación presentamos algunos de estos problemas y cómo podrían ser tratados en un trabajo futuro.

### 5.3.1. Detección de canales no cerrados

Uno de los problemas a los que se enfrenta un programador de Go al lidiar con canales respecto al *leak* de memoria y recursos es que si se crean canales continuamente y no se cierran luego de usarlos cuando ya puede asegurarse que no van a volver a ser usados en ninguna ejecución futura, se pueden acumular *buffers* abiertos que ocupan espacio en la memoria dinámica y pueden llegar a drenar recursos del sistema en términos generales. La herramienta **Gong** es capaz de establecer un punto en la ejecución para el cual se puede asegurar que un canal nunca va a ser cerrado debido a que puede calcular cuan rápido funciona la propiedad de pérdida de memoria sobre las llamadas recursivas (la cota $k$). Esto se puede lograr utilizando las barbas, con las cuales se puede validar un predicado que indique que el canal será eventualmente cerrado por un determinado hilo de ejecución. Si un canal es "olvidado" (lo que es equivalente a decir que se está reconociendo un patrón de comunicación que ya fue visto y por lo tanto no se debe agregar un nuevo estado al LTS) y no se valida la barba de cierre eventual de ese canal, se puede establecer que ese canal no va a cerrarse nunca. Si además es posible evaluar (quizás con otra barba) que ese canal no volverá a ser utilizado por ningúna rutina viva al momento de constatar la propiedad de

pérdida de memoria entonces es posible adaptar **Gong** para indicar la presencia de *leaks* respecto a la memoria utilizada por los *buffers* de los canales y sugerir un numero de línea a partir del cual el canal se podría cerrar sin afectar el funcionamiento del programa.

### 5.3.2. Potencialmente infinitas rutinas inmortales (*goroutine leak*)

Otro *leak* de recursos importante tiene que ver con *goroutinas* que ejecutan infinitamente. Si un programa puede generar potencialmente infinitas rutinas de este tipo entonces eventualmente se irán consumiendo recursos del scheduler ralentizando el funcionamiento del sistema en general. Es posible que la generación infinita de rutinas infinitas sea algo deseado. Sin embargo, hay una gran cantidad de casos para los cuales esto no es así y la generación infinita de rutinas que no terminan de ejecutar es de hecho un *leak* de recursos del sistema muy serio (por ejemplo, en un servidor que nunca cierra las sesiones de los pedidos que atiende). Por lo tanto, tiene sentido alertar al usuario que este caso existe para que luego él pueda decidir si tomar en cuenta la advertencia o ignorarla a sabiendas que el comportamiento descripto es el deseado.

Se podrían identificar potenciales infinitas rutinas recorriendo el LTS generado para el tipo de un programa constatando si en algún ciclo del LTS se genera una composición en paralelo de una invocación a una función infinita. Una función es infinita si existe al menos una posible ejecución de ésta que no termina. Una forma de ver que la función no termina es, por ejemplo, ver si en algún punto de su ejecución ésta se llama recursivamente con los mismos parámetros.

### 5.3.3. Equivalencia entre canales y primitivas de sincronización

Tomando en cuenta la polémica sobre la conveniencia o no de la utilización de primitivas de sincronización por sobre la utilización de canales [6] y considerando que las primitivas de sincronización también cuentan con un uso extendido, se considera como un posible trabajo futuro extender el modelo de tipo sesión junto con su implementación a las primitivas de sincronización como un caso particular de sincronización utilizando canales. En la figura 5.3 tenemos un programa con la estructura típica de un *lock* de exclusión mutua (*mutex*) con la salvedad de que está implementado utilizando canales. Si se corre primero **Dingo Hunter**[1] y luego se corre **Gong** sobre el resultado podemos verificar que el código de la figura está libre de *deadlocks*. Si por ejemplo, agregamos en la línea 17 (*unlock*) un condicional que puede hacer que en un posible flujo de ejecución no se libere el *lock*, **Gong** detecta el *deadlock*. Si se puede establecer una correspondencia entre un *mutex* y un canal con *buffer* con capacidad igual a uno y a las operaciones de obtener el lock y devolverlo como recibir y mandar sobre el canal, se podría adaptar la herramienta para interpretar cada lock como si en verdad fuera un canal.

## 5.4. Recomendación de correcciones

Una vez que se identifica la instrucción que genera una violación de la propiedad de *safety* o *liveness* se pueden intentar correcciones automáticas estándar para ver si alguna logra corregir el programa. Una posibilidad puede ser, en caso de una falla de

[1] por los problemas comentados en la sección 5.2 el resultado de correr Dingo-Hunter debe ser corregido a mano para que el ejemplo funcione

```
1. package main
2.
3. import (
4.         "fmt"
5.         "time"
6. )
7.
8. type ValueUpdater struct {
9.         mutex  chan int
10.        global int
11. }
12.
13. func (vu *ValueUpdater) UpdateGlobal(i int) {
14.        <-vu.mutex
15.        vu.global += i
16.        fmt.Println(vu.global)
17.        vu.mutex <- 1
18. }
19.
20. func main() {
21.        mutex := make(chan int, 1)
22.        vu := ValueUpdater{}
23.        vu.mutex = mutex
24.        go update(&vu)
25.        go update(&vu)
26.        vu.mutex <- 1
27.        time.Sleep(1 * time.Second)
28. }
29.
30. func update(vu *ValueUpdater) {
31.        vu.UpdateGlobal(1)
32. }
```

*Fig. 5.3:* Mutex implementado con un canal con *buffer* de capacidad 1

*liveness*, aumentar la capacidad del canal que genera el problema. Otra opción puede ser simplemente eliminar la instrucción problemática. Opciones más complejas pueden incluir tratar de ubicar si existen operaciones reciprocas que ya hayan sincronizado con la misma operación que genera la falla en un punto previo de la ejecución e intentar multiplicarlas esperando que puedan arreglar el problema. Para cada posible solución, puede generarse un nuevo LTS con los cambios propuestos y constatar luego si se corrige el programa (al menos en términos de las propiedades analizadas). Para cada corrección que convierta al programa nuevamente en *live* o *safe* se puede sugerir al usuario las modificaciones pertinentes para que este decida si las acepta o no.

## 5.5. Tiempo de ejecución

Las modificaciones realizadas en **Gong** tuvieron un impacto en el tiempo de ejecución de la herramienta. En la figura 5.4 se puede observar el resultado de medir tiempos de ejecución mediante el programa `time` de bash[2] para distintos programas utilizados como ejemplos para popl17 que acompañaron la publicación en [8]. El tiempo de ejecución de la verificación de las propiedades aumenta en todos los casos aunque en distinta proporción. Para el programa Altbit, el tiempo de ejecución es aproximadamente 5 veces más. Para Prime Sieve (la criba de eratóstenes) es casi 30 veces más lento. Para la versión asincrónica de la secuencia de Fibonacci es sólo un 50 % más lenta mientras que para el ForSelect de la figura 4.1 el tiempo de ejecución es el doble.

Las modificaciones realizadas agregan tiempo de cómputo en la medida en que se concatena información sobre la historia de ejecución del programa, sino las herramientas funcionan igual. En ese sentido, la principal hipótesis sobre cómo varía el tiempo de ejecución tiene que ver con cuán largas pueden ser las trazas generadas para cada programa. La cantidad de estados del LTS generado junto con su topología es un factor a tener en cuenta debido a que indican la máxima cantidad de sincronizaciones que pueden ocurrir hasta que la ejecución simbólica se termine. Otro factor es la cantidad de funciones definidas en el contexto del programa, ya que potencialmente, dentro de cada estado del LTS pueden invocarse estas funciones agregando la información de la invocación en la traza.

Por otro lado, el aumento respecto al tiempo de ejecución también puede llegar a explicarse por cómo funciona la concatenación de *strings* en Haskell, la cual tiene un tiempo de ejecución lineal respecto al tamaño de los dos *strings* que se van a concatenar. En este sentido, se intentó reemplazar el tipo *string* del campo de los GoTypes de la figura 3.4 por el tipo DList de la librería Data.DList de Haskell basado en listas enlazadas que brinda operaciones de concatenación en tiempo constante esperando observar una mejoría en el tiempo de ejecución. Sin embargo, esta modificación no pudo ser llevada a cabo debido a que al cambiar el tipo de este campo de los GoType estos no pueden instanciarse como derivación del tipo *Alpha*, algo que se necesita para poder hacer *binds* de nombres de canales o funciones.//

Queda pendiente como trabajo futuro intentar mejorar el tiempo de ejecución de **Gong**. Para programas de mayor complejidad, éste se vuelve un fuerte limintante respecto a su uso.

[2] man time

```
Altbit:                          Fibonacci-asynch:
Kstates: 52                      K-States: 24
        Original:   0m0.252s             Original:   0m19.869s
        Modificado: 0m1.354s             Modificado: 0m29.985s

PrimeSieve:                      ForSelect con deadlock:
K-States: 13                     Kstates: 5
        Original:   0m0.166s             Original:   0m0.019s
        Modificado: 0m4.387s             Modificado: 0m0.041s
```

*Fig. 5.4:* Tiempos de ejecución para los ejemplos popl17 en [8]

## Bibliografía


[1] `https://tour.golang.org/concurrency/1`.

[2] `https://gobyexample.com/channels`.

[3] 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.

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

[5] `https://blog.golang.org/8years`.

[6] `https://www.jtolio.com/2016/03/go-channels-are-bad-and-you-should-feel-bad/`.

[7] Nicolas Dilley and Julien Lange. An empirical study of messaging passing concurrency in go projects. 2019.

[8] 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.

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

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

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

[12] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[13] Joachim Parrow. An introduction to the $\pi$-calculus. In *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001.

[14] Robin Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

[15] Hans Hüttel, Ivan Lanese, Vasco T Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, et al. Foundations of session types and behavioural contracts. *ACM Computing Surveys (CSUR)*, 49(1):3, 2016.

[16] Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, (1):157–171, 1986.

[17] Naoki Kobayashi. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In *IFIP International Conference on Theoretical Computer Science*, pages 365–389. Springer, 2000.

[18] http://simonjf.com/2016/05/28/session-type-implementations.html.

[19] Julian Rathke, Vladimiro Sassone, and Paweł Sobociński. Semantic barbs and biorthogonality. In *International Conference on Foundations of Software Science and Computational Structures*, pages 302–316. Springer, 2007.

[20] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

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

[22] https://github.com/nickng/gong/blob/master/Liveness.hs#L99.

[23] https://github.com/nickng/gong/blob/master/Safety.hs#L84.

[24] https://github.com/nickng/gong/blob/master/SymbolicSem.hs#L92.

[25] https://github.com/DamiFur/dingo-hunter.

[26] https://github.com/DamiFur/gong.

[27] https://github.com/nickng/gong/blob/master/SymbolicSem.hs#L23.

[28] http://hackage.haskell.org/package/unbound-0.5.1.1/docs/Unbound-LocallyNameless.html#

[29] http://hackage.haskell.org/package/unbound-0.5.1.1/docs/Unbound-LocallyNameless.html.

[30] https://github.com/DamiFur/gong/blob/Trace/GoTypes.hs#L57.