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

Estudio Empírico del Uso de Funciones de Concurrencia en Proyectos Rust

Tesis de Licenciatura en Ciencias de la Computación

David Temnyk

Gustavo Torrecilla

Director: Hernán Melgratti

Buenos Aires, 2021

ESTUDIO EMPÍRICO DEL USO DE FUNCIONES DE CONCURRENCIA EN PROYECTOS RUST

Este trabajo se propone aportar evidencia empírica sobre el uso de funciones de concurrencia, principalmente envío de mensajes en canales, en aplicaciones Rust, un lenguaje de programación que provee mecanismos novedosos para garantizar *memory safety* y *thread safety*. En líneas generales, el trabajo sigue el enfoque metodológico propuesto en [1] para caracterizar aspectos de concurrencia en aplicaciones Go. Para esto, identificamos un repositorio significativo de proyectos Rust, diseñamos un mecanismo para obtener los proyectos de interés del mismo, desarrollamos una herramienta para generar el AST (*Abstract Syntax Tree*) a partir del código y computar descriptores de interés, tales como frecuencia en el uso de primitivas de comunicación, peso en el uso de las mismas y mecanismos de comunicación predominantes en los proyectos, y utilizamos dicha herramienta para realizar la experimentación sobre los proyectos seleccionados, presentando un análisis descriptivo de los resultados.

Palabras claves: Rust, Concurrencia, Comunicación, Canal, AST.

AN EMPIRICAL STUDY OF CONCURRENCY FUNCTIONS USAGE IN RUST PROJECTS

This work is aimed at providing empirical evidence about the usage of concurrency functions, mainly message-passing communication, on Rust applications. Rust is a programming language that features novel mechanisms for guaranteeing properties such as *memory safety* and *thread safety*. Our methodological approach goes along the lines of the one proposed in [1] for the study of concurrency aspects in Go applications, which can be summarised as follows: 1) we identified a significant public repository of Rust projects, 2) we devised a mechanism for automatically obtaining relevant projects, 3) we developed a tool to generate an AST (Abstract Syntax Tree) representation of programs out of their code, 4) we computed selected statistical descriptors about the usage of communication primitives, e.g., frequency, average weight, predominant communication mechanisms and 5) we analyzed the obtained results.

Keywords: Rust, Concurrency, Communication, Channel, AST.

Índice general

1. INTRODUCCIÓN

A menudo el diseño y desarrollo de técnicas de análisis de programas o modelos formales de lenguajes de programación limitan su alcance a fragmentos de un lenguaje, por ejemplo, restringiendo el conjunto de funciones o primitivas que pueden ser utilizadas o prohibiendo algunos patrones de código [2]. Tales restricciones pueden deberse a cuestiones de simplicidad técnica, limitaciones del enfoque o cuestiones de decidibilidad y eficiencia en el análisis. Suele ser generalmente desafiante proveer una técnica de análisis para propiedades interesantes de programas que sea *sound* y *complete*, y a su vez, pueda ser aplicada a la totalidad de un lenguaje de programación de producción industrial.

En consecuencia, queda siempre en el aire la pregunta sobre la relevancia de una determinada técnica que no soporta completamente a la totalidad de un lenguaje. Es decir, saber si el fragmento considerado es suficientemente amplio como para incluir a buena parte de las aplicaciones que se escriben usualmente en tal lenguaje. Es por este motivo que resulta interesante contar con una caracterización de las aplicaciones que se escriben típicamente en un determinado lenguaje de programación. Varias técnicas fueron propuestas recientemente para probar propiedades sobre aplicaciones concurrentes que se comunican a través del intercambio de mensajes [3][4][5][6], muchas de las cuales consideran sólo algunos modelos de comunicación (comunicación sincrónica vs. asincrónica), limitan la cantidad de canales o el modo en el que los mismos se crean y utilizan. Es por ello que es de interés disponer de una descripción cuantitativa del uso que se realiza de tales funciones en lenguajes concretos.

Este trabajo se propone aportar evidencia empírica sobre el uso de funciones de concurrencia en aplicaciones Rust, un lenguaje de programación que provee mecanismos novedosos para garantizar *memory safety* y *thread safety* [7], sobre el cual se realizaron varios trabajos recientes centrados en dichos aspectos [8][9]. Para esto, identificamos un repositorio significativo de proyectos Rust públicos y analizamos la frecuencia de uso de dichas funciones, el peso en el uso de las mismas y los mecanismos de comunicación (librerías) y topologías de concurrencia predominantes en dichos proyectos.

2. CONTEXTO

2.1. El lenguaje Rust

Rust [10] es un lenguaje de programación multiparadigma, open-source, que pone énfasis en *performance* y *reliability*. Su sistema de tipos y modelo de *ownership* [11] garantiza tanto *memory safety* como *thread safety* sin necesidad de contar con un Garbage Collector [12]. El lenguaje nació en 2006 como un proyecto personal de Graydon Hoare, empleado de Mozilla. En 2009 dicha empresa se convirtió en sponsor del proyecto y en 2015 se lanzó la primera versión estable del lenguaje.

Uno de los principales objetivos de Rust es poder manejar la programación concurrente de manera eficiente y segura [13]. En la mayoría de los sistemas operativos de hoy en día, el código de un programa en ejecución corre en un proceso y el SO maneja múltiples procesos a la vez. Dentro de un programa, puede haber componentes independientes que corren en simultáneo. Los encargados de correr estos componentes independientes son los *threads*.

Dividir un programa en varios threads puede mejorar la performance ya que el programa puede realizar varias tareas al mismo tiempo, pero también agrega cierto grado de complejidad. Como los threads ejecutan de forma simultánea, no existen garantía sobre el orden en que se ejecutarán las diferentes partes del código en los distintos threads. Esto puede llevar a problemas como:

- Race conditions, donde los threads acceden a datos o recursos en un orden inconsistente.
- Deadlocks, en donde dos threads están esperando cada uno por los recursos que está utilizando el otro, evitando que ambos puedan avanzar.
- Bugs que ocurren solo en ciertas circunstancias y son por lo tanto difíciles de reproducir y corregir de manera confiable.

Los lenguajes de programación implementan threads de distinta manera. Muchos sistemas operativos proveen APIs para crear threads. Este modelo donde un lenguaje llama a la API del SO para crear threads suele ser llamado 1:1, lo cual significa que hay un 1 thread del SO por 1 thread del lenguaje. [14]

Algunos lenguajes de programación proveen su propia implementación de threads. Estos se denominan *green threads* y son ejecutados en el contexto de un número diferente de threads del SO. Por esta razón, este modelo suele ser llamado M:N, donde hay M green threads y N threads del sistema operativo, siendo M y N no necesariamente iguales.

Cada modelo tiene sus propias ventajas y *trade-offs*. La librería estándar de Rust provee una implementación del modelo 1:1 por cuestiones de performance, pero existen librerías que proveen implementaciones del modelo M:N.

Una forma muy popular de garantizar *safe concurrency* es el envío de mensajes, donde distintos threads o actores se comunican entre sí enviándose mensajes que contienen datos [15]. Existe un slogan para esto, el cual es utilizado por el lenguaje Go: “Do not communicate by sharing memory; instead, share memory by communicating.” [16]

Una de las herramientas principales con las que cuenta Rust para lograr concurrencia

a través del envío de mensajes son los **Channels** o canales. Los mismos pueden pensarse como canales de agua, tal como un río, donde todo objeto que se coloque en ellos, viaja por la corriente río abajo. Un canal tiene 2 extremos: un transmisor (o emisor) y un receptor. Una parte del programa llama métodos del transmisor para enviar mensajes con datos y otra parte del programa utiliza al receptor para recibirlos. Se dice que un canal está **cerrado** si el transmisor o el receptor fue *dropeado*.

Rust provee una implementación de canales en su librería estándar, la cual se denomina `mpsc::channel`, donde mpsc significa "multiple producer, single consumer". Es decir, un canal puede tener muchos emisores pero un solo receptor. Existen, sin embargo, otras librerías que proveen implementaciones de canales con múltiples receptores también.

En el presente trabajo analizaremos 3 librerías distintas en Rust, cada una con su propia implementación de canales: la ya mencionada librería estándar [17], `Crossbeam` [18] y `Flume` [19]. La primera fue elegida por ser la default, mientras que las otras 2 se seleccionaron por ser librerías populares que vienen a resolver problemas presentes en la librería estándar y que además ofrecen una mejor performance.

En Rust cada librería provee sus propias funciones para crear canales. Todo canal posee un buffer asociado donde se guardan los mensajes enviados hasta que sean consumidos por algún thread. Dicho buffer puede ser acotado o no, y el tamaño del mismo se denomina capacidad del canal. Si la capacidad es 0, entonces el canal es sincrónico: cualquier operación que se ejecute sobre él provocará que el thread se bloquee hasta que otro thread ejecute la operación de sincronización complementaria. Por otro lado, si la capacidad del canal es estrictamente mayor a 0, el comportamiento del canal depende del estado del buffer. Tanto el envío de mensajes en un canal no lleno como la recepción de los mismos en un canal no vacío son operaciones no bloqueantes. Es decir que en estos casos el canal se comporta de forma asincrónica. Sin embargo, tratar de enviar un mensaje en un canal lleno o de recibir un mensaje en un canal vacío provocarán que el thread se bloquee, comportándose entonces de manera sincrónica. Para el caso de que el buffer no esté acotado, se trata de un modelo asincrónico donde un thread jamás puede bloquearse al enviar mensajes, sino que solo puede hacerlo si intenta recibir mensajes en un canal vacío.

En `MPSC`, se utilizan los métodos *channel* y *sync_channel* para la creación de canales. El primero permite la creación de un canal de capacidad infinita, retornando una tupla con un Sender (utilizado para enviar mensajes) y un Receiver (utilizado para recibirlos). Todas las operaciones de escritura son no bloqueantes, mientras que las operaciones de lectura sólo son bloqueantes si el canal está vacío. En el caso de sync_channel, el mismo toma como parámetro la capacidad del canal y retorna una tupla con SyncSender y Receiver. Los métodos para crear canales en `Crossbeam` y `Flume` se llaman *bounded* y *unbounded*, siendo los mismos análogos a los métodos sync_channel y channel de `MPSC`, respectivamente. La única diferencia es que `Crossbeam` y `Flume` utilizan el mismo tipo Sender para ambos (es decir, no establecen ninguna distinción entre Sender y SyncSender).

Los canales en Rust se consideran desconectados cuando alguno de sus extremos es dropeado, es decir, se libera la memoria asociada al mismo. Si un Sender trata de enviar un mensaje sobre un canal desconectado, va a recibir un resultado de error. No obstante, si un Receiver trata de recibir un mensaje de un canal desconectado, podrá hacerlo siempre y cuando existan todavía mensajes en el buffer. Una vez agotados los mismos, se recibirá un error.

En las 3 librerías es posible clonar el extremo emisor utilizando la función *clone*. Ésta es la única manera de tener múltiples productores sobre un mismo canal. Sin embargo,

sólo `Crossbeam` y `Flume` permiten clonar el extremo receptor. La librería estándar de Rust no ofrece esta posibilidad.

Las 3 librerías permiten iterar sobre el buffer de mensajes. Existen 2 maneras de realizar esto. La primera es con la función *iter*, la cual retorna un iterador bloqueante. Mientras haya mensajes en el buffer, los mismos serán retornados. Una vez agotados, el iterador se va a quedar esperando que lleguen nuevos mensajes. Esta operación termina cuando el canal es desconectado. La segunda manera es con la función *try_iter*, la cual retorna un iterador no bloqueante. El comportamiento es similar al anterior con la diferencia de que la operación termina al quedar vacío el buffer.

`Crossbeam` y `Flume` cuentan con la operación select, la cual permite esperar sobre múltiples operaciones de comunicación. Un select está compuesto de distintas ramas, cada una de las cuales posee una guarda con una acción de comunicación. El select se bloquea hasta que alguna de sus ramas se pueda sincronizar y luego ejecuta el bloque correspondiente a ese rama. En el caso de existir al mismo tiempo dos o más ramas con las que se pueda sincronizar, entonces elegirá una de forma no determinística. Es posible también especificar una rama default para evitar que se bloquee el select cuando no existen operaciones con las que pueda sincronizarse.

En Rust, todos los canales tienen un tipo asociado, el cual queda determinado en el momento de su creación y no puede ser modificado. No existen restricciones sobre dicho tipo, por lo que es posible crear "canales de canales".

Para el manejo de memoria, Rust utiliza un sistema de *ownership* [20]. El mismo consiste en un conjunto de reglas que se chequean en tiempo de compilación. Este sistema es una de las características principales de Rust, que permite brindar garantías de *memory safety*, sin la necesidad de un *garbage collector*, reduciendo de forma considerable el overhead en tiempo de ejecución. Es importante entender el mismo con claridad, ya que tiene fuertes implicaciones en el resto del lenguaje. Las reglas de ownership son las siguientes:

- Cada valor tiene una variable llamada *owner*.
- Solo puede haber un owner al mismo tiempo.
- Cuando el owner deja de pertenecer al *scope*, el valor se *dropea*.

El scope de un elemento es el conjunto de instrucciones dentro de un programa para el cual dicho elemento se encuentra definido. En principio, toda variable es válida desde el momento en el que es creada hasta el final del bloque actual. Cuando una variable deja de ser válida, Rust se encarga de dropearla, liberando la memoria asociada a la misma.

Estas restricciones de ownership son las que impiden utilizar un mismo Sender o Receiver en distintos threads al mismo tiempo y por lo tanto fuerzan a utilizar las funciones específicas de clonación.

2.2. Trabajos relacionados

En 2019, Dilley y Lange [1] realizaron un estudio sobre el uso de primitivas de concurrencia en el lenguaje Go. En el mismo, buscan respuesta a 4 preguntas específicas:

- ¿Con qué frecuencia se utilizan operaciones de envío de mensajes en Go?
- ¿Cómo se distribuye la concurrencia en proyectos Go?

- ¿Qué tan común es el uso de envío de mensajes asincrónicos en Go?
- ¿Qué topologías de concurrencia se utilizan en Go?

El lenguaje Go provee lo que se conoce como *goroutines*. Las mismas son lightweight threads administrados por el scheduler de Go, los cuales se caracterizan por consumir un mínimo de recursos. El lenguaje también provee canales como tipo estándar para facilitar la comunicación entre distintas goroutines. En esto se diferencia de Rust, el cual utiliza threads del SO, limitando la cantidad de threads que se puede utilizar en comparación con las goroutines pero ofreciendo una mayor performance en su uso.

En Go se utiliza la primitiva *make* para crear canales. La misma toma como parámetro el tipo de canal y opcionalmente una capacidad. En caso de no especificarse capacidad, la misma toma 0 como valor default. Los canales de Go son bidireccionales, por lo que pueden ser utilizados tanto para enviar como para recibir mensajes, a diferencia de Rust, donde existe un extremo dedicado a cada operación. Además de esto, no existen canales con capacidad infinita en Go.

Cualquier goroutine con una referencia válida a un canal puede cerrar el mismo para indicar que ya no se van a enviar más valores. Cualquier intento posterior de enviar un mensaje por un canal cerrado provocará un panic failure. Lo mismo ocurrirá si se intenta cerrar un canal ya cerrado. Sin embargo, es posible continuar recibiendo mensajes de un canal cerrado. Si el buffer aún no está vacío, entonces se van ir consumiendo los mensajes restantes, y cuando el buffer se vacía, se va a recibir el valor default del tipo de canal (por ej: para canales de enteros, se va a recibir el valor 0).

En Go se puede utilizar la función *range* para iterar sobre los mensajes de un canal hasta que el mismo sea cerrado y también existe la función *select*, la cual le permite a una goroutine esperar sobre múltiples operaciones de comunicación.

Una propiedad interesante de Go es que los canales se pueden guardar y compartir con cualquier otra parte del programa, lo cual nos permite la creación de canales de canales. Un uso común de esta propiedad es para implementar demultiplexación segura y en paralelo [21].

Tras analizar 900 proyectos de Github, Dilley y Lange llegaron a las siguientes conclusiones:

- El 76 % de los proyectos analizados emplean canales. La primitiva receive es la más utilizada y el promedio de primitivas por canal es bajo, sugiriendo que los canales solo se utilizan para protocolos de sincronización simples.
- Menos de la mitad de los paquetes de proyectos Go analizados contienen features de concurrencia, mientras que con respecto a la cantidad de archivos esto representa solo el 20 %. Los archivos con features de concurrencia suelen tener una mayor tamaño que los que son solo secuenciales.
- Los canales sincrónicos son los que se utilizan más comúnmente (61 %) mientras que los canales asincrónicos se utilizan con una capacidad conocida estáticamente, la cual es menor o igual a 5 en el 75 % de los casos.
- El 58 % de los proyectos analizados incluyen creaciones de threads en for loops y la mayoría de los proyectos (87 %) utilizan un número acotado de canales.

3. METODOLOGÍA

En líneas generales el trabajo sigue el enfoque metodológico propuesto en [1], donde se caracterizan aspectos de concurrencia en aplicaciones Go.

Primero, identificamos un repositorio significativo de proyectos en Rust. Para esto, consideramos aquellos proyectos disponibles en GitHub, cuya cantidad actual aproximada es de 60.000. Los ordenamos en función de su cantidad de estrellas de manera descendente y tomamos para este estudio los primeros 900, cuyo rango de estrellas oscila entre 19.300 y 169.

Diseñamos un mecanismo para obtener los proyectos de interés a través de la API de GitHub. Para esto, desarrollamos un script en Python que se conecta con la API y retorna la lista de proyectos, la cual posteriormente analizamos para filtrar aquellos que no incluyen código Rust a pesar de estar catalogados como tales o que poseen el propio código de las librerías de canales que nos interesan.

Desarrollamos una herramienta que permite analizar el código Rust. Inicialmente consideramos distintas librerías para la obtención del AST (Árbol Sintáctico Abstracto) de los archivos *.rs*, entre ellas *syntex_syntax*. Sin embargo, debido a la rápida evolución del lenguaje Rust y a la falta de mantenimiento de la librería, la misma presentaba muchos errores. Por este motivo, optamos por utilizar una funcionalidad experimental del compilador *rustc* (versión 1.42.0-nightly), la cual se encarga de exportar el AST en formato JSON para cada archivo *.rs*. Posteriormente, cargamos dicha representación en Python y la parseamos a clases para proceder a su análisis.

Implementamos un programa Python, el cual recorre el AST de todos los archivos de cada proyecto y computa las siguientes ocurrencias de features:

- Funciones de creación de canales de las diferentes librerías:
  - channel y sync_channel para `MPSC`.
  - bounded, unbounded, tick, never y after para `Crossbeam`.
  - bounded y unbounded para `Flume`.

  A su vez, se almacena la capacidad de cada canal, siempre que la misma pueda ser conocida estáticamente.
- Funciones sobre Senders/SyncSenders y Receivers, tales como send, recv, iter, etc.
- Dropeo de Senders/SyncSenders y Receivers.
- Funciones sobre Selects. En este punto, hay que aclarar que por cuestiones de complejidad no estamos considerando la utilización de macros como *select!*, sino que solo contabilizamos las funciones que se ejecutan sobre el struct Select.
- Creación de threads mediante el método *spawn* de la librería estándar. Además, se consideran los casos especiales en los que los threads (y también los canales) son creados en un for loop.
- Aliasing de canales dentro de for loops.

- Almacenamiento de Senders/SyncSenders o Receivers en vectores o arrays.

Las métricas computadas por cada proyecto son almacenadas en archivos JSON. Posteriormente, se cargan estos datos en un analizador para obtener resultados globales.

Para comparar la intensidad en el uso de funciones de concurrencia en proyectos de distinto tamaño y estructura, seguimos el mismo enfoque de Dilley y Lange sobre presentar las mediciones de forma relativa a la cantidad de líneas físicas de código (PLOC). Para esto, utilizamos el comando CLOC [22] (v1.88), el cual descarta líneas vacías y comentarios. Dado un proyecto P, escribimos $|P|$ como su *concurrent size*, es decir, la suma de las PLOC de todos los archivos .rs que contienen al menos un feature de concurrencia enumerado en esta sección. Matemáticamente, $|P| = \sum_{f \in F(P)} kPLOC(f)$, donde $F(P)$ es el conjunto de archivos en P que contienen al menos un feature de concurrencia.

4. IMPLEMENTACIÓN

4.1. Obtención de código fuente

Para obtener los proyectos, utilizamos la API que expone GitHub, filtrando aquellos que utilizan código Rust y ordenándolos de manera descendente en función de su cantidad de estrellas (`https://api.github.com/search/repositories?q=+language:rust&sort=stars&order=desc&per_page=10`). En cada búsqueda obtenemos 10 proyectos por página, removemos aquellos que son de documentación o que contienen la propia implementación del lenguaje Rust y realizamos el análisis de los mismos antes de pasar a los siguientes 10. Finalmente, repetimos este proceso varias veces hasta alcanzar el número estipulado de 900 proyectos.

4.2. Parsing

Para analizar el código Rust en busca de funciones de concurrencia, utilizamos una funcionalidad experimental de rustc, la cual se encarga de exportar el AST de un archivo .rs en formato JSON.

```
rustc -Z ast-json <<file>>
```

A continuación mostraremos un ejemplo simple de un código Rust y el resultado obtenido luego de correr el comando. En el Listing 4.1 se puede ver un ejemplo en el que se utiliza la función de creación de canales de capacidad infinita provisto por la librería estándar.

```
use std::sync::mpsc;

fn channel_creation(){
    let (tx1, rx1) = mpsc::channel();
}
```

Listing 4.1: Ejemplo de código Rust

En la figura 4.1 se puede ver la representación visual del AST para la instrucción *use* de este código. Es importante mencionar que tanto en esta imagen como en las posteriores que utilicemos, vamos a estar omitiendo propiedades del AST que no resulten relevantes.

La figura 4.2 corresponde únicamente a la instrucción `mpsc::channel()`. Por cuestiones de simplicidad, no se incluyen en la misma ni la definición de la función ni tampoco la asignación de las variables.

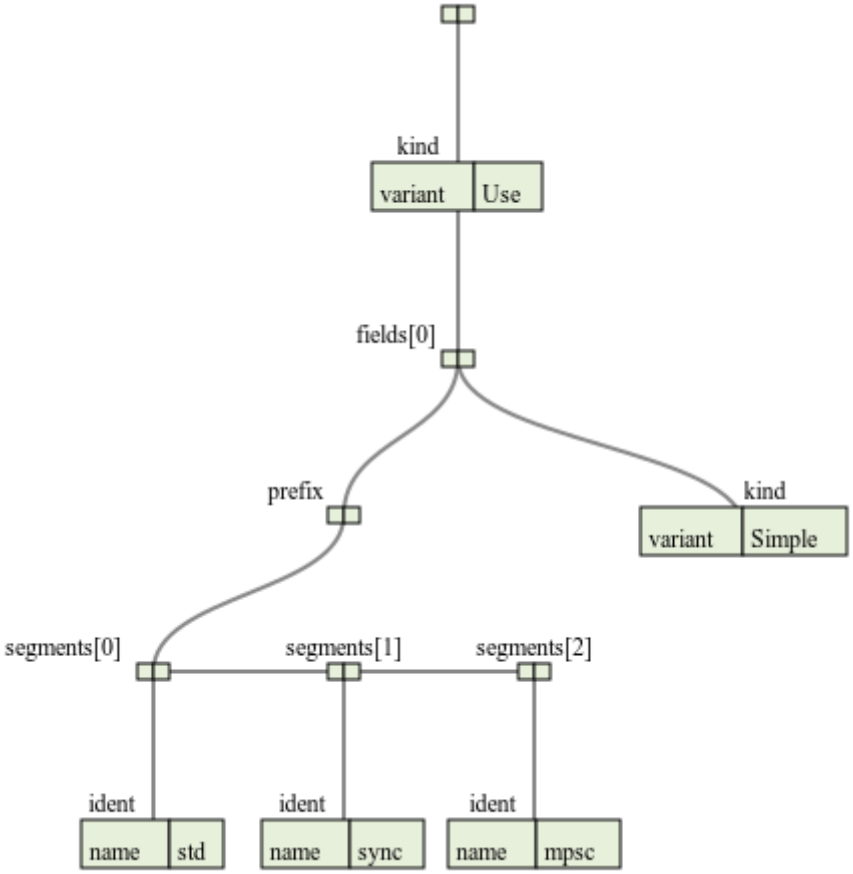

Fig. 4.1: Representación visual de AST para la instrucción use del Listing 4.1

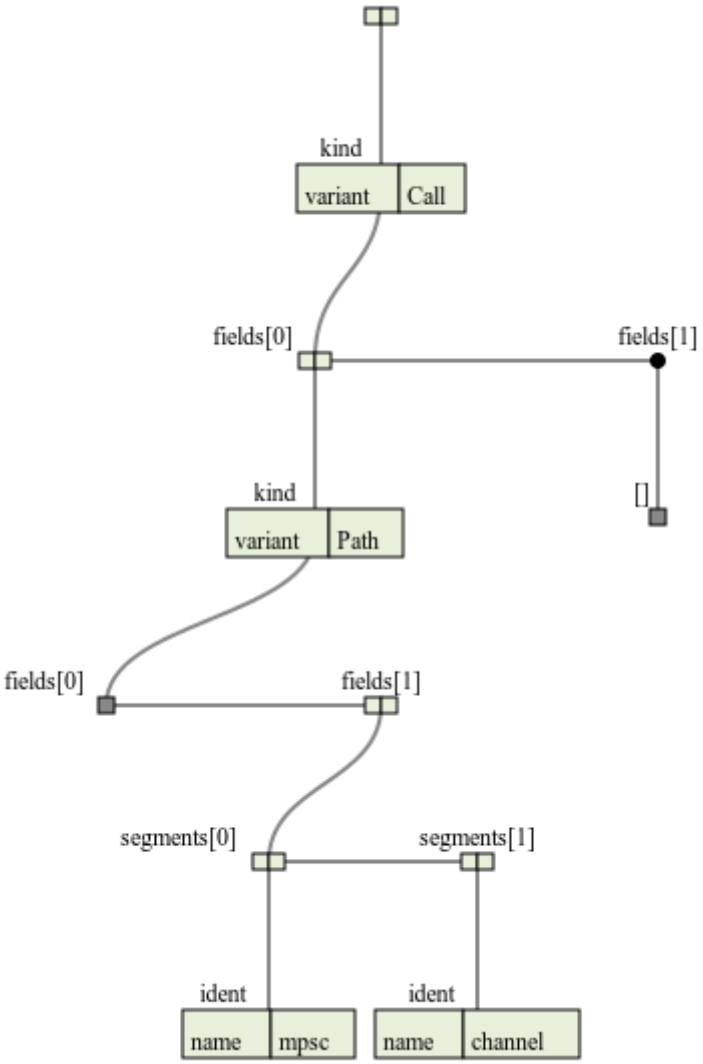

Fig. 4.2: Representación visual de AST para creación de canal

Una vez obtenida la representación JSON del AST, leemos la misma desde un programa Python y lo deserializamos a clases que fueron creadas específicamente para esto. A continuación explicaremos sobre los elementos más importantes del AST.

El elemento raíz del AST es un *Crate*. El mismo contiene principalmente un *Módulo*, que a su vez está compuesto por una lista de *Items*. Cada ítem representa un elemento del AST junto con metadata asociada y posee un tipo específico, denominado *ItemKind*. Este último tiene una propiedad *variant* que sirve para discriminar los distintos ítems, lo cual resulta útil para el propósito de este trabajo. Por ejemplo, en la Figura 4.1, vemos un ItemKind cuyo variant es *Use*. Cada ItemKind posee una taxonomía distinta, la cual se expresa como un array de *fields*.

Los tipos de ItemKind que existen actualmente son:

- ExternCrate: ítem asociado al *extern crate*. Ej: `extern crate foo`
- Use: declaración de *use*. Ej: `use foo::bar`

- Static: declaración de variables estática. Ej: `static FOO: i32 = 42`
- Const: declaración de constante. Ej: `const FOO: i32 = 42`
- Fn: declaración de función. Ej: `fn foo(bar: usize) -> usize { .. }`
- Mod: declaración de módulo. Ej: `mod foo { .. }`
- ForeignMod: módulo externo. Ej: `extern {}`
- GlobalAsm: inline assembly a nivel de módulo. Ej: `global_asm!(..)`
- TyAlias: un alias de tipo. Ej: `type Foo = Bar<u8>`
- Enum: definición de tipo enumerado. Ej: `enum Foo<A, B> { C<A>, D<B> }`
- Struct: definición de struct. Ej: `struct Foo<A> { x: A }`
- Union: definición de unión. Ej: `union Foo<A, B> { x: A, y: B }`
- Trait: declaración de trait. Ej: `trait Foo { .. }`
- TraitAlias: alias de trait. Ej: `trait Foo = Bar + Quux`
- Impl: implementación de un struct. Ej: `impl<A> Foo<A> { .. }`
- MacCall: invocación de macro. Ej: `foo!(..)`
- MacroDef: definición de macro. Ej: `macro_rules! A { .. }`

Siguiendo con el Listing 4.1, la declaración de la función *channel_creation* se representa con un Ítem cuyo ItemKind es *Fn*. Entre los fields de éste hay un *Block*, el cual representa el cuerpo de la función. Dicho Block contiene una lista de *Statements*, donde cada uno representa una instrucción de la función. Los Statements tienen, a su vez, un *StatementKind* asociado, el cual puede ser de distintos tipos. En nuestro ejemplo, el tipo del mismo es *Local* y se utiliza para representar la creación (y opcionalmente asignación) de una variable local. Dentro del Local, existe un elemento denominado *Expr*, el cual representa expresiones de código RUST parseadas. Cada Expr tiene asociado también un *ExprKind* con su respectiva lista de fields.

Algunos de los tipos de ExprKind más relevantes en nuestro trabajo son:

- Assign: Asignaciones de variables.
- Call: Llamadas a funciones estáticas o de creación de instancias.
- MethodCall: Llamadas a métodos de objetos.
- While: Ciclo con una condición booleana.
- ForLoop: Ciclo for tradicional.
- Loop: Ciclo sin condición.
- Path: Secuencia de identificadores que se utilizan para representar un nombre. Ej: `std::sync::mpsc::channel()`

En la Figura 4.2, se observa el ExprKind asociado a la expresión `mpsc::channel()`, el cual es un *Call*. Si recorremos los fields del mismo, terminaremos llegando a otro Expr, cuyo ExprKind asociado es de tipo *Path*. Este último es el que nos sirve para identificar cuál es la función que se está llamando y en combinación con la lista de Use también nos permite saber cuál es la librería a la que pertenece. En la Figura 4.3, se puede observar una representación visual completa de los elementos del AST involucrados en la función.

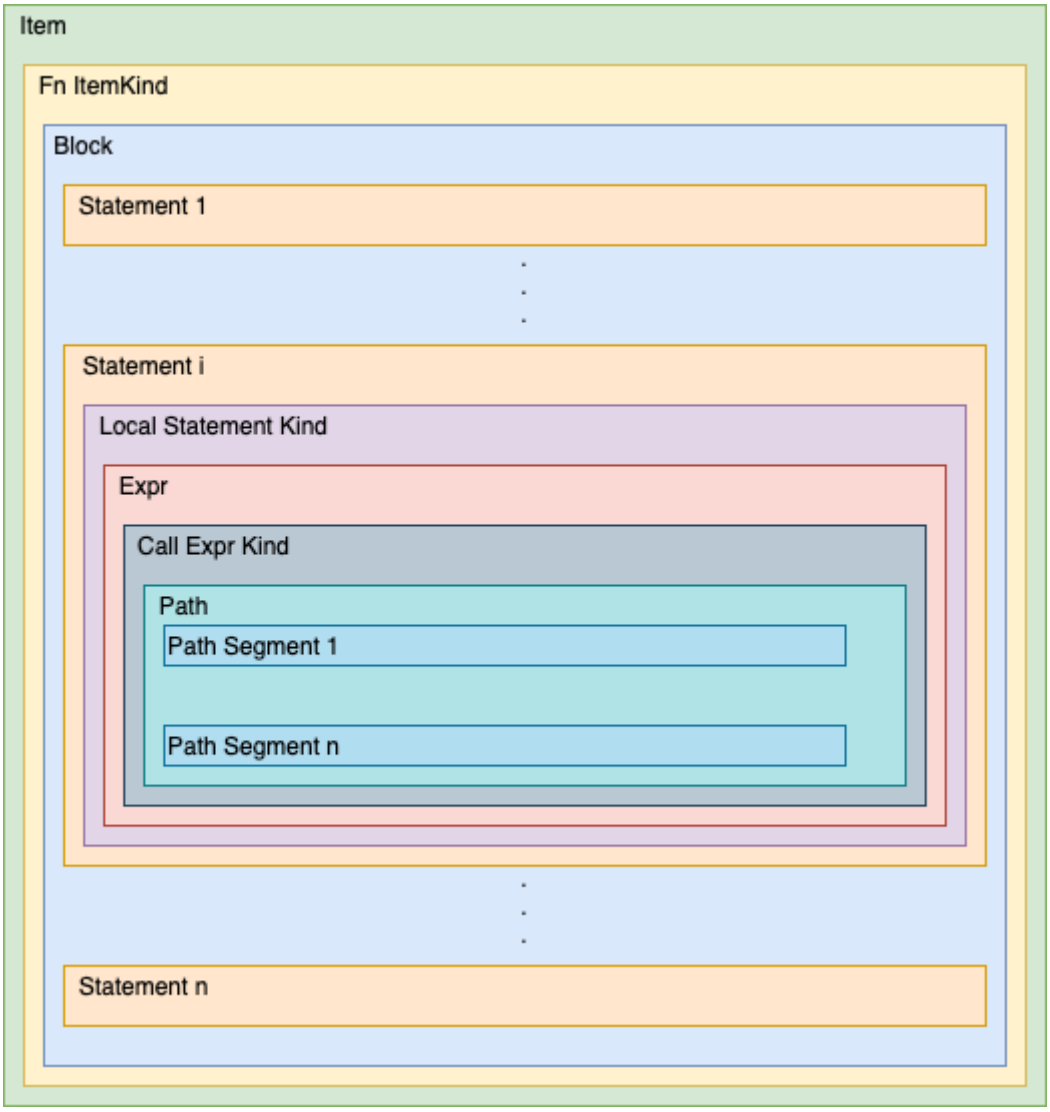

Fig. 4.3: Representación visual de la Fn del Listing 4.1

4.3. AST Visitor

Luego de parsear el AST, utilizamos un Visitor para recorrer el mismo y recolectar los datos necesarios para el análisis. Hacemos una primera pasada sobre el AST de cada archivo para recolectar información sobre los Use, y una segunda pasada donde identificamos los features descriptos en la sección de Metodología. La primera pasada nos sirve

para saber si las llamadas a funciones corresponden a las librerías que nos interesan para el análisis y además nos permite discriminar específicamente sobre cuál de estas librerías se está realizando el llamado. Esto es importante dado que en muchos casos las librerías comparten los nombres de las funciones.

```
use crossbeam_channel::unbounded;

fn channel_creation(){
    let channel = unbounded();
}
```

Listing 4.2: Ejemplo de `Crossbeam`

```
use flume::unbounded;

fn channel_creation(){
    let channel = unbounded();
}
```

Listing 4.3: Ejemplo de `Flume`

En los Listings 4.2 y 4.3 se puede ver como la instrucción para crear un canal no acotado tiene el mismo nombre tanto para `Crossbeam` como para `Flume`.

Los use pueden ser declarados en el mismo archivo junto con el código o en otros archivos, siendo declarados como públicos. Un patrón frecuente en los proyectos Rust es el de generar uno o más archivos *lib.rs* y colocar allí todos los pub use generales del proyecto. En esta primera pasada de nuestra herramienta, sólo nos enfocamos en los pub use declarados en dichos archivos así como los use específicos de cada archivo a analizar.

La segunda pasada es la que se encarga puntualmente de recolectar los datos de interés para nuestro análisis. A continuación, describiremos el trabajo realizado sobre cada uno de los features anteriormente mencionados.

4.3.1. Funciones de creación de canales

Visitamos los StatementKind de tipo Local y nos fijamos que en la inicialización de los mismos exista un ExprKind de tipo Call. Luego, obtenemos el nombre de la función utilizada analizando el Path del mismo y contrastamos esta información con la obtenida a partir de los Use para saber si se trata de una de las funciones de creación de canales. En caso afirmativo, contabilizamos la creación de canal para la librería que corresponda y guardamos en un diccionario la información sobre las variables a las cuales se están asignando los Senders/SyncSenders y Receivers generados por dicha función. Las funciones de creación de canales retornan en su mayoría una tupla. En los Listings 4.2 y 4.3 podemos observar como se asigna dicha tupla a una variable. Sin embargo, en Rust también es posible realizar un Tuple Unpack, bindeando cada valor de la tupla a una variable distinta, tal cual se muestra en el Listing 4.1. Ambos casos son tenidos en cuenta a la hora de almacenar la información.

La librería `Crossbeam` provee además 3 canales especiales, llamados tick, never y after, los cuales tienen un manejo ligeramente distinto al resto, ya que las funciones de creación de canales sólo retornan un Receiver.

Además de los StatementKind de tipo Local, también se analizan los ExprKind de tipo

Assign. Los mismos se diferencian de los Local en que los primeros se utilizan para crear variables locales y opcionalmente asignarlas, mientras que los últimos son solo para asignar variables ya creadas. El trabajo realizado en este caso es análogo al ya mencionado.

En Rust, también es posible clonar los extremos de un canal ya existente. Para esto se invoca el método clone del Sender/SyncSender o Receiver que se desea clonar, siempre y cuando esto sea posible (recordar que los canales de `MPSC` no permiten clonar los Receivers). A nivel AST, se realiza un análisis sobre los Local StatementKind y Assign ExprKind, en los cuales se busca el ExprKind de tipo MethodCall correspondiente al clone de alguno de los Senders/SyncSenders o Receivers (o tuplas de los mismos) almacenados en nuestro diccionario. En la sección Funciones sobre Senders y Receivers explicamos un poco más en detalle como es el análisis general de los MethodCall.

Para cada variable de nuestro diccionario se almacena la siguiente información:

- Nombre de la variable.
  Dentro de cada Local existe un elemento denominado Pat, el cual representa la variable que se está declarando. La misma posee un PatKind, el cual puede ser de distintos tipos. Si se trata de una tupla o de uno de los Receivers especiales de `Crossbeam`, el PatKind es de tipo Ident. Este último es el que contiene el nombre de la variable. Por otro lado, si se trata de un Tuple Unpack, entonces el PatKind es de tipo Tuple. Este último no es más que un vector de Pat (en nuestro caso, siempre de tamaño 2), donde el PatKind de cada uno es de tipo Ident. Por otro lado, para el caso de los Assign ExprKind, los mismos poseen un elemento de tipo Expr, el cual representa al lado izquierdo de la asignación. Dicho elemento posee un ExprKind de tipo Path, por lo que es posible obtener el nombre de la variable a partir de él. Es importante aclarar que no se puede realizar un Tuple Unpack en los Assign ExprKind.

- Tipo de la variable.
  En todos los casos en que se analiza una función de creación de canales y su asignación en un Tuple Unpack, el tipo de la primera variable se almacena como Sender, salvo que la función utilizada sea el sync_channel de `MPSC`, en cuyo caso se almacena como SyncSender. Por otro lado, el tipo de la segunda variable se almacena siempre como Receiver. Para el caso en que se esté asignando una tupla, el tipo de la misma va a ser Tupla. Más adelante, veremos también que las variables que representan colecciones de canales se almacenan también con tipo Vector o Array. Por último, a las variables en las que se guardan los Receivers devueltos por las funciones tick, never y after de `Crossbeam` les asignamos a cada una un tipo específico distinto.

- Tipo de dato del canal, si es posible.
  Dentro de cada Local existe un elemento denominado Ty, el cual representa al tipo de la variable que se está creando. Dado que la anotación de tipo es opcional, este elemento no siempre va a estar en el AST. En los casos en los que sí esté, es posible recorrer la estructura del mismo para extraer la información sobre cuál es el tipo de datos que se va a enviar en el canal. El principal uso que se le da a esta información es analizar si se está en presencia de un canal de canales.

- Capacidad del canal, si es posible.
  Si la función de creación de canal es la sync_channel de `MPSC` o la bounded de `Crossbeam` o `Flume`, entonces al analizar el Call ExprKind respectivo, se obtiene su

lista de argumentos y en caso de que el primer elemento sea un Expr que represente un literal (es decir, su ExprKind asociado sea de tipo Lit), se extrae el valor numérico del mismo, ya que éste representa la capacidad del canal. Es importante mencionar que con este análisis, sólo estamos considerando aquellas capacidades que estén definidas estáticamente. Un caso aparte son las variables de tipo After, Tick y Never, ya que las primeras 2 siempre van a estar asociadas a canales de capacidad 1 y la última de capacidad 0.

- Librería a la que corresponde.
  Utilizando la información obtenida de los use, es posible identificar cuál es la librería asociada a cada función de creación de canales.

- Id del canal.
  Es un id creado por nosotros para identificar unívocamente cuáles son los extremos asociados a un mismo canal.

- Diccionario de métodos ejecutados.
  En él guardamos aquellos métodos que sean invocados sobre la variable, así como la cantidad de cada uno.

En todos los casos, cuando lo que se está analizando es un clone, entonces se copia toda la información de la variable original, con excepción del nombre y el diccionario de métodos ejecutados.

4.3.2. Funciones sobre Senders y Receivers

Visitamos los ExprKind de tipo MethodCall, tomamos el primer parámetro de su lista de argumentos, validamos que su ExprKind sea de tipo Path, extraemos el nombre del mismo, el cual representa a la variable a la cual se le está ejecutando el método, verificamos si dicho nombre está en nuestro diccionario de variables y en caso afirmativo, guardamos la ocurrencia del método en el diccionario de métodos ejecutados de la variable. En el Listing 4.4, se puede apreciar un ejemplo de una función en la que se crea un canal de `MPSC`, se envía un número sobre el Sender y se obtiene el mismo desde el Receiver. En las Figuras 4.4 y 4.5 se encuentran los AST asociados a las instrucciones send y recv.

Además, si el método a ejecutar es un **send**, verificamos si el elemento a enviar (segundo parámetro de la lista de argumentos) corresponde alguna de las variables que tenemos almacenadas. En caso afirmativo, almacenamos también el tipo de dicha variable, de acuerdo a lo definido en la sección anterior. Esto es de gran utilidad a la hora de identificar canales de canales.

```
fn mpsc_channel(){
    let (tx, rx) = std::sync::mpsc::channel();
    tx.send(1);
    rx.recv();
}
```

Listing 4.4: Ejemplo de Send y Receive en `MPSC`

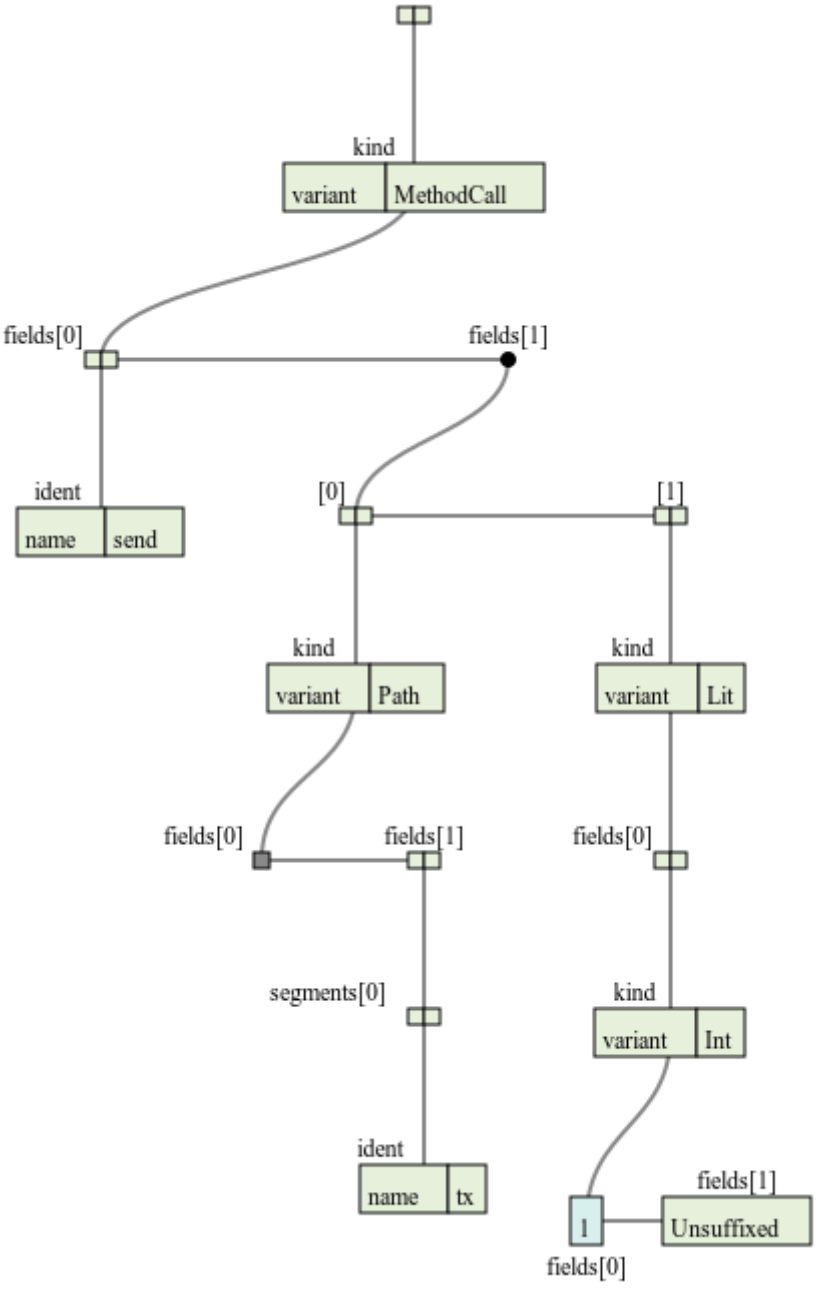

Fig. 4.4: Representación visual de AST sobre instrucción send

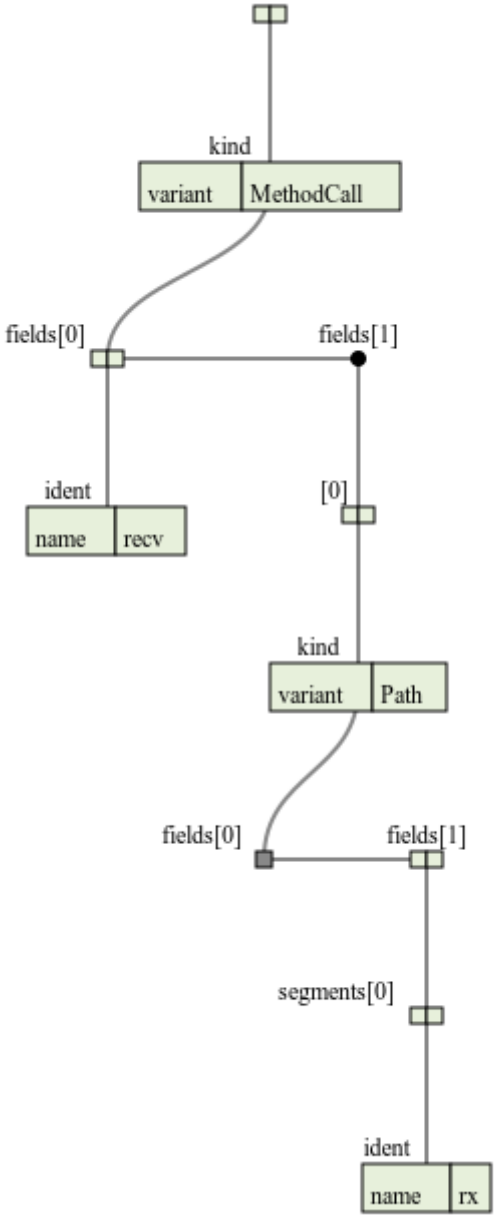

Fig. 4.5: Representación visual de AST sobre instrucción recv

4.3.3. Dropeo de Senders y Receivers

Visitamos los ExprKind de tipo Call, obtenemos el nombre de la función utilizada analizando el Path del mismo y verificamos si la misma corresponde a un drop. En caso afirmativo, tomamos la lista de argumentos, analizamos el nombre del primer elemento y validamos que corresponda a una variable de nuestro diccionario. Si la validación es positiva, guardamos entonces la ocurrencia del drop en el diccionario de métodos ejecutados de la variable. En el Listing 4.5 tenemos un ejemplo de un drop y en la figura 4.6, el AST correspondiente.

```
fn channel_drop(){
    let (tx1, rx1) = std::sync::mpsc::channel();
    drop(tx1);
}
```

Listing 4.5: Ejemplo de drop

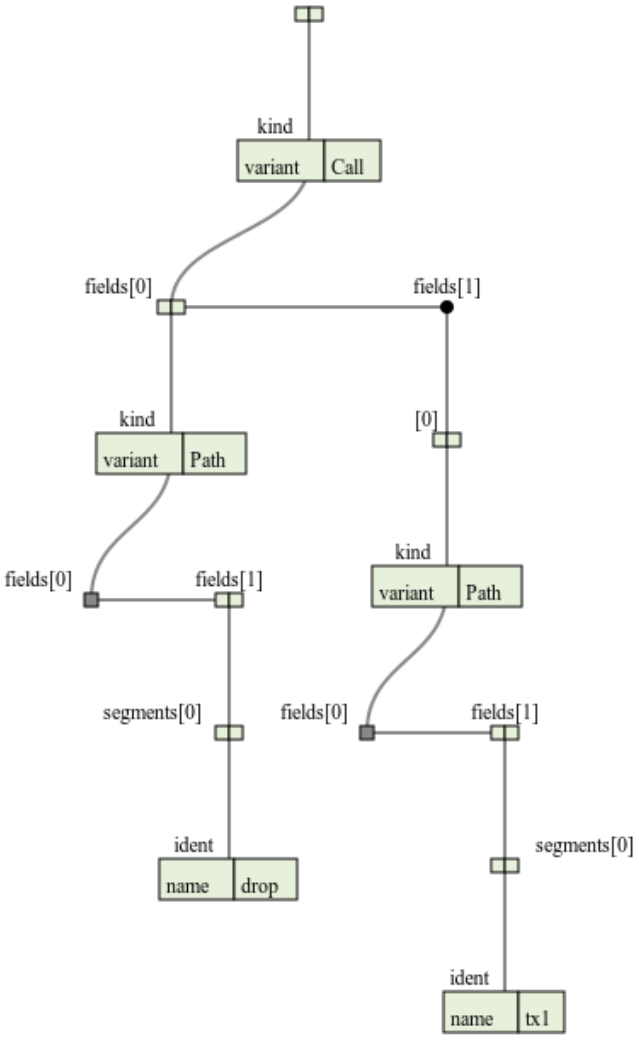

Fig. 4.6: Representación visual de AST sobre instrucción drop

4.3.4. Funciones de Selects

Para la creación de Selects en `Crossbeam` y Selectors en `Flume`, el procedimiento es análogo al de las funciones de creación de canales. Visitamos los StatementKind de tipo Local y nos fijamos que en la inicialización de los mismos exista un ExprKind de tipo Call. Luego, obtenemos el nombre de la función utilizada analizando el Path del mismo y contrastamos esta información con la obtenida a partir de los Use para saber si se trata de una de las funciones de creación de Selects/Selectors. En caso afirmativo, guardamos en el diccionario la información sobre la variable a la cual se le está realizándole la asignación, poniéndole tipo Select. En el Listing 4.6, hay un ejemplo de código Rust sobre la utilización de un Select y en la Figura 4.7 está el AST de la creación del mismo.

```
fn crossbeam_select_example(s1:Sender<i32>,r1:Receiver<i32>){
    let mut sel = Select::new();
    let oper1 = sel.recv(&r1);
    let oper2 = sel.send(&s1);
    let oper = sel.select();
    match oper.index() {
        i if i == oper1 => oper.recv(&r1),
        i if i == oper2 => oper.send(&s1, 10),
        _ => unreachable!(),
    }
}
```

Listing 4.6: Ejemplo de Select en `Crossbeam`

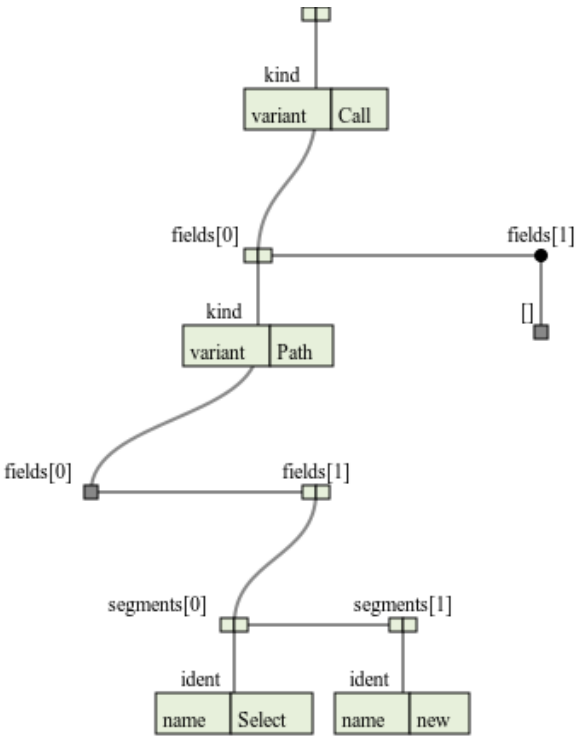

Fig. 4.7: Representación visual de AST sobre creación de Select

Para las funciones que se ejecutan sobre los Selects, el procedimiento es análogo al de las funciones sobre Senders y Receivers. Visitamos los ExprKind de tipo MethodCall, tomamos el primer parámetro de su lista de argumentos, validamos que su ExprKind sea de tipo Path, extraemos el nombre del mismo, verificamos si dicho nombre está en nuestro diccionario de variables con tipo Select y en caso afirmativo, guardamos la ocurrencia del método en el diccionario de métodos ejecutados de la variable. En el Listing 4.8 está el AST correspondiente a la instrucción recv del código Rust mostrado en el Listing 4.6.

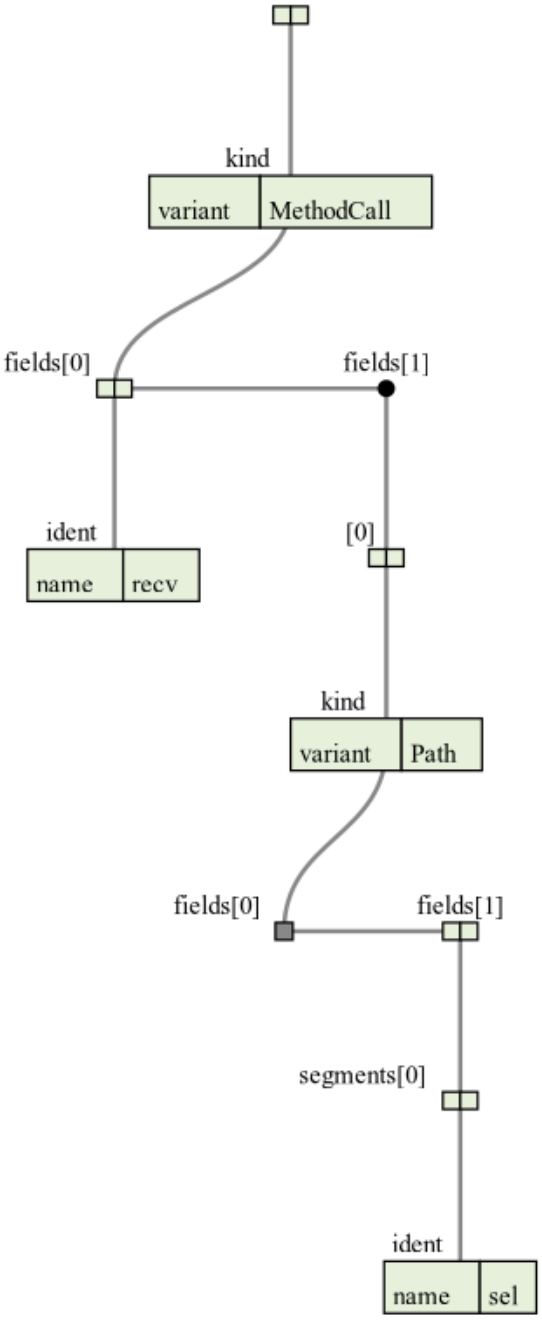

Fig. 4.8: Representación visual de AST sobre función recv de Select

En el caso de `Flume`, una práctica frecuente es utilizar una misma instrucción para crear un Selector, definir cuáles van a ser sus ramas de ejecución y realizar la elección. Esto es posible porque las funciones send y recv de Selector (utilizadas para definir las ramas) retornan Self, a diferencia de los Select de `Crossbeam`. Este comportamiento es tenido en cuenta durante el análisis, en el cual definimos variables frescas en el diccionario para los Selectors y analizamos la invocación de los métodos de forma recursiva. En el Listing 4.7 hay un ejemplo de un Selector en `Flume`, en el cual se utilizan todas las invocaciones a métodos de forma anidada y en la Figura 4.9 está el AST correspondiente a la creación del Selector y a la invocación de los 3 métodos.

```
fn select(tx1: Sender<i32>, rx1: Receiver<i32>){
    Selector::new()
        .send(&tx1, 43, |n| println!("Sended {:?}", n))
        .recv(&rx1, |n| println!("Received {:?}", n))
        .wait();
}
```

Listing 4.7: Ejemplo de Selector en `Flume`

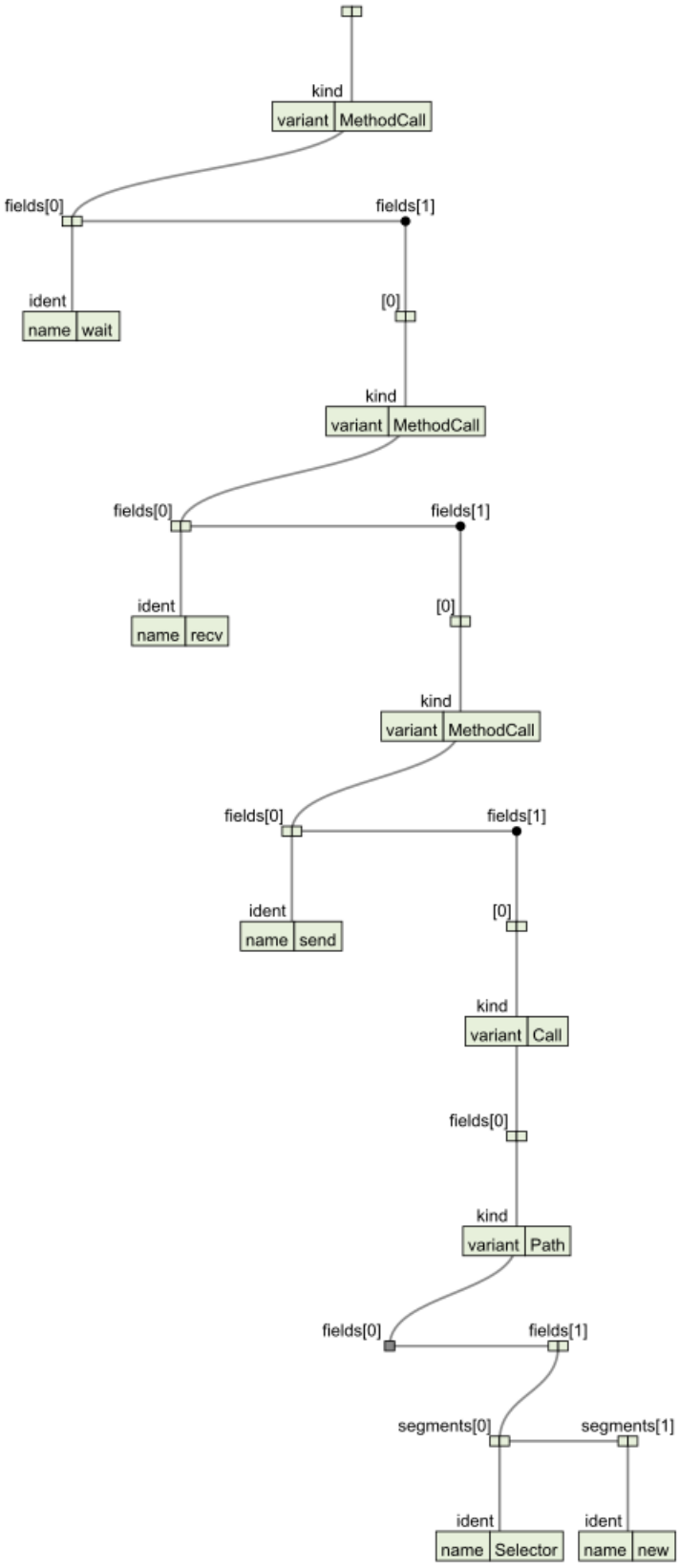

Fig. 4.9: Representación visual de AST sobre Selector en flume

4.3.5. Creación de Threads

Para la creación de Threads, realizamos un trabajo similar al de los dropeos de canales. Visitamos los ExprKind de tipo Call, obtenemos el nombre de la función utilizada analizando el Path del mismo y verificamos si la misma corresponde a un spawn de la librería estándar. En caso afirmativo, contabilizamos la creación del Thread. En el Listings 4.8 y la Figura 4.10 hay un ejemplo sobre un spawn y su correspondiente AST.

```
fn spawn_in_loop() {
    let handle = thread::spawn(move || {
        let (tx1, rx1) = crossbeam_channel::unbounded();
    });
}
```

Listing 4.8: Ejemplo de spawn

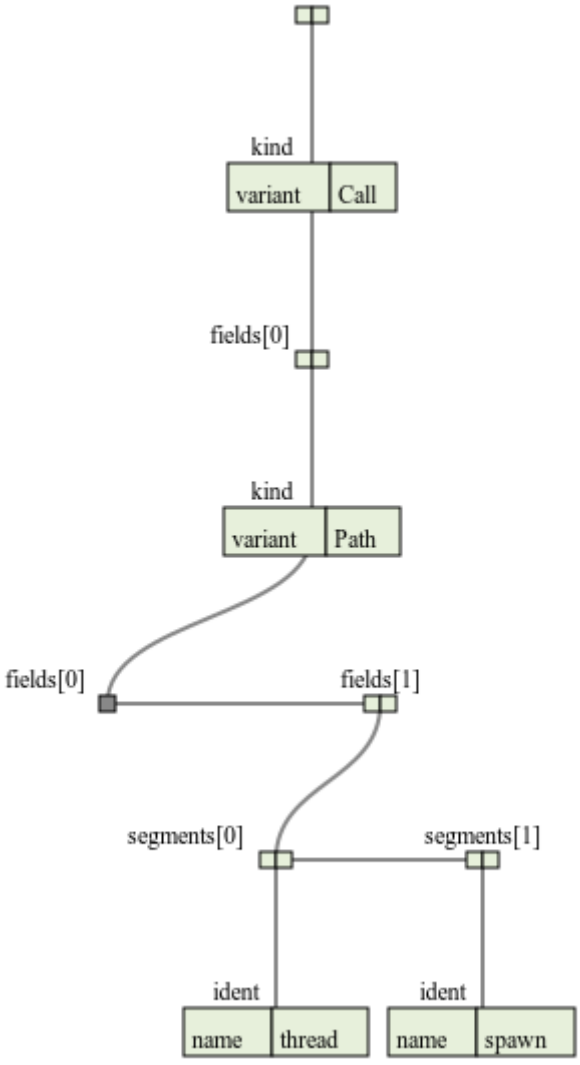

Fig. 4.10: Representación visual de AST sobre spawn

4.3.6. Aliasing y Creación de Threads y Canales dentro de Ciclos

Un caso especial de interés para nuestro trabajo es la contabilización de la creación de canales y threads dentro de ciclos, así como también la utilización de aliasing sobre los Senders/SyncSenders o Receivers. Tal como explicamos anteriormente, existen en principio 3 tipos de ciclos en Rust: Loop, ForLoop y While. En los Listings 4.9, 4.10 y 4.11 hay ejemplos de cada uno. A nivel AST, cada ciclo se representa con un ExprKind distinto.

Los 3 tipos de ExprKind contienen un *Block*, el cual representa el cuerpo del ciclo. En la sección de Parsing explicamos un poco sobre la composición del Block. El análisis continúa dentro del Block hasta llegar a los ExprKind que nos interesan. En todos los casos, cada vez que "entramos" a un ciclo, nos guardamos en una lista el scope correspondiente (While, Loop o ForLoop), y al "salir" del ciclo, quitamos el scope de la lista. Una vez localizadas las funciones de creación de canales y threads, guardamos los llamados en el scope en el que fueron hechos. Este manejo de scopes nos permite tener en cuenta ciclos anidados.

En el caso de los ExprKind de tipo ForLoop, nos interesa determinar la cantidad de iteraciones de los mismos. Para esto, analizamos el Expr del interior del ForLoop, chequeamos que su ExprKind sea de tipo Range (el cual representa un rango) y en caso afirmativo, validamos que ambos extremos se correspondan con literales de tipo Int. De verificarse estas condiciones, procedemeos a guardar la diferencia entre ambos extremos. Es importante destacar que esto sólo es posible en casos en que el rango de iteraciones esté definido de manera estática. En la Figura 4.11 se puede apreciar el fragmento de AST correspondiente al for loop del Listing 4.10.

Para el aliasing de canales, chequeamos los Local y Assign ExprKind en los cuales la parte derecha se corresponda con un Sender o Receiver existente y además corroboramos que la lista de scopes no sea vacía (es decir, que estemos adentro de un ciclo). En caso de que esto suceda, guardamos la ocurrencia del aliasing asociado al scope correspondiente. Es importante que mencionar que si bien estamos utilizando el término aliasing, debido al modelo de memoria que maneja Rust, este tipo de asignaciones produce un cambio de ownership sobre el valor de la variable, por lo que la única referencia válida al mismo pasa a ser la de la nueva variable.

```
fn loop_assign_aliasing(){
    let (tx1, rx1) = mpsc::channel();
    let mut tx: mpsc::Sender<i32>;
    loop {
        tx = tx1;
        tx.send(k).unwrap();
    }
}
```

Listing 4.9: Ejemplo de aliasing en loop

```
fn mpsc_for(){
    for i in 1..33 {
        let (tx1, rx1) = mpsc::channel();
        tx1.send(i).unwrap();
    }
}
```

Listing 4.10: Ejemplo de creación de canal en for

```
fn spawn_while(){
    while true {
        let handle = thread::spawn(move || {
            let (tx1, rx1) = crossbeam_channel::unbounded();
        });
    }
}
```

Listing 4.11: Ejemplo de creación de thread en while

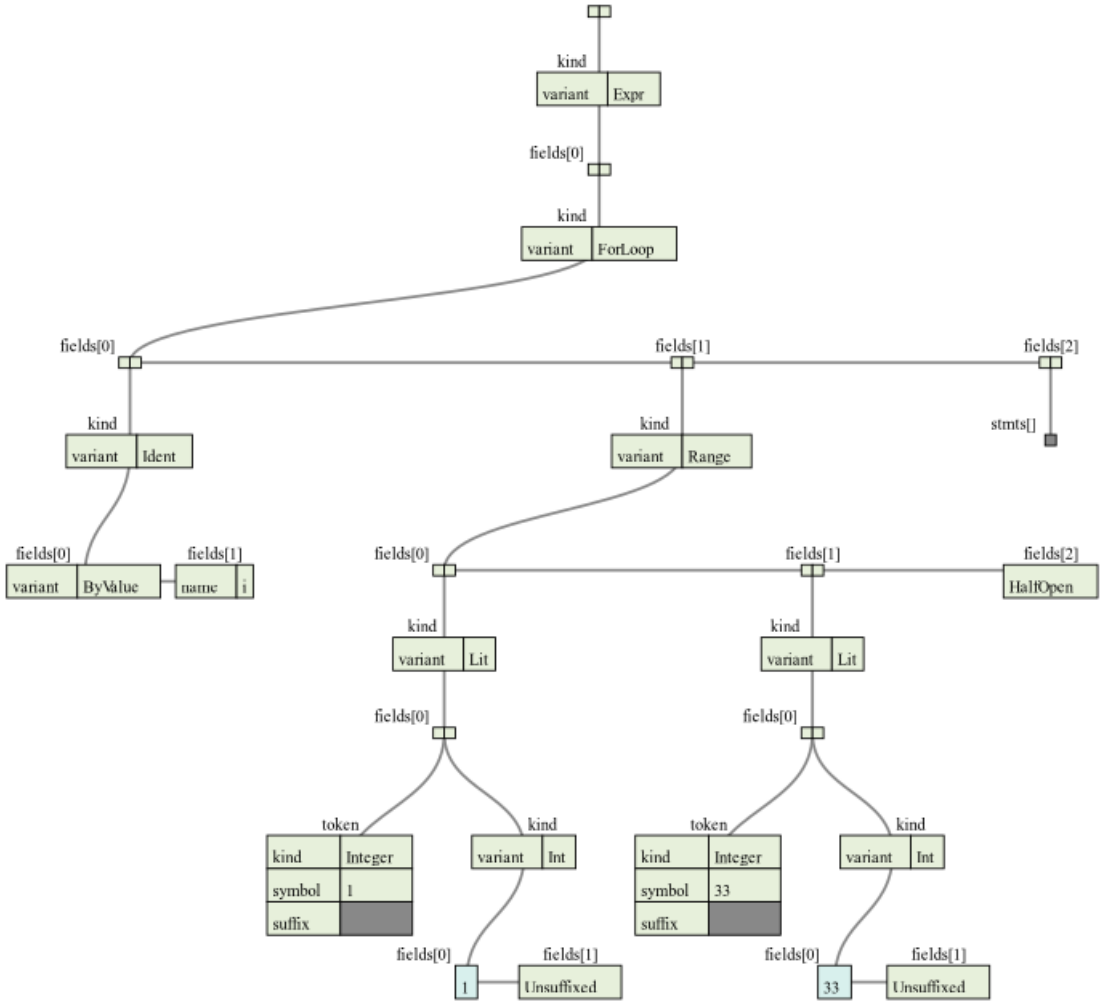

Fig. 4.11: Representación visual de AST sobre creación de canal en for

4.3.7. Colecciones de canales

Otro caso de interés es cuando los canales se almacenan en colecciones. En el contexto de esta tesis, tendremos en cuenta únicamente Arrays y Vectores. En ambos casos, visitamos los Local o Assign ExprKind, tal cual explicamos en otras secciones y analizamos el lado derecho de la asignación. Para los Arrays, el ExprKind de la derecha debe ser de tipo Array, su lista de valores debe ser no vacía y alguno de estos valores debe corresponderse

con una variable de nuestro diccionario. Por otro lado, para los Vectores, el ExprKind de la derecha debe ser de tipo Call, el nombre de la función tiene que ser *into_vec* (la misma corresponde al código generado por la macro *vec!*), y su lista de valores debe cumplir las mismas condiciones que para los Arrays. En caso de verificarse estas condiciones, guardamos en nuestro diccionario una nueva variable con tipo Array o Vector, según corresponda, junto con el tipo de los elementos que contiene (Senders, Receivers, etc).

A la hora de contabilizar las funciones sobre canales, también estamos teniendo en cuenta las colecciones. Por ejemplo: si en los ExprKind de tipo MethodCall, el primer argumento de la lista se corresponde con un ExprKind de tipo Index y este último contiene ExprKind de tipo Path, el cual se corresponde a una variable de tipo Vector o Array en nuestro diccionario, entonces definimos en dicho diccionario una variable fresca y contamos la ocurrencia del llamado a la función. Mismo tratamiento se da para cuando hay un aliasing de un canal almacenado en una colección. En el Listing 4.12 hay un ejemplo de creación de vector de SyncSender y de envío de un mensaje sobre el elemento de la primera posición. En la Figura 4.12 se puede apreciar el fragmento de AST correspondiente a la creación y en la Figura 4.13, el correspondiente al indexado y al posterior envío de mensaje.

```
fn mpsc_sync_sender_vec(){
    let mut vec;
    let (sync_sender, rx) = mpsc::sync_channel(10);
    vec = vec![sync_sender];
    vec[0].send(1);
}
```

Listing 4.12: Ejemplo de vector de SyncSender

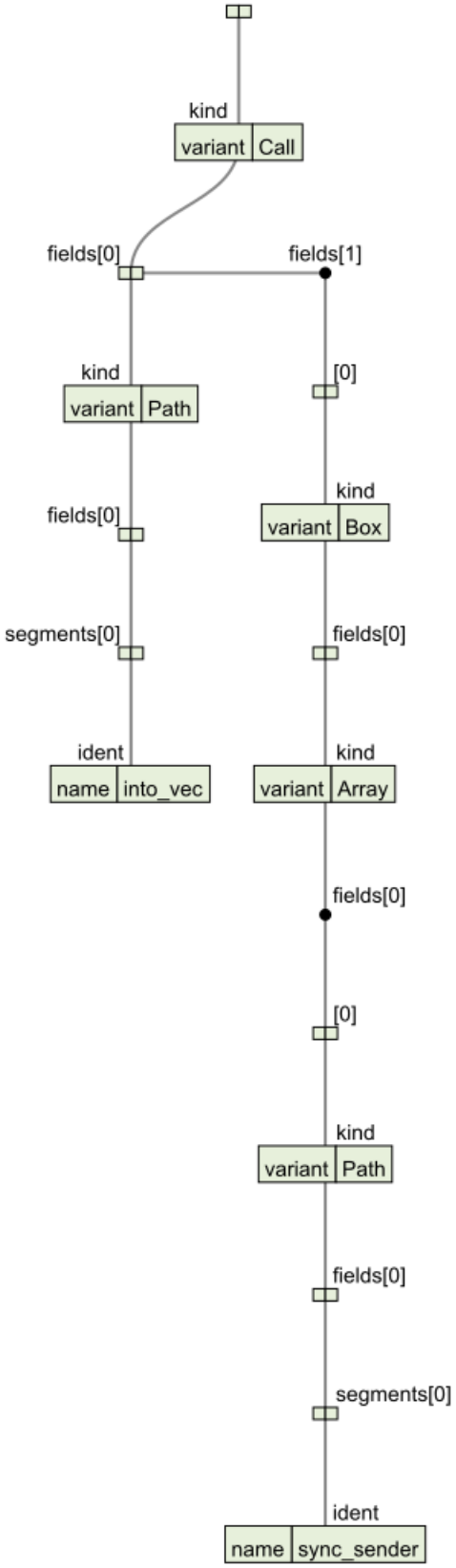

Fig. 4.12: Representación visual de AST sobre creación de vector de SyncSender

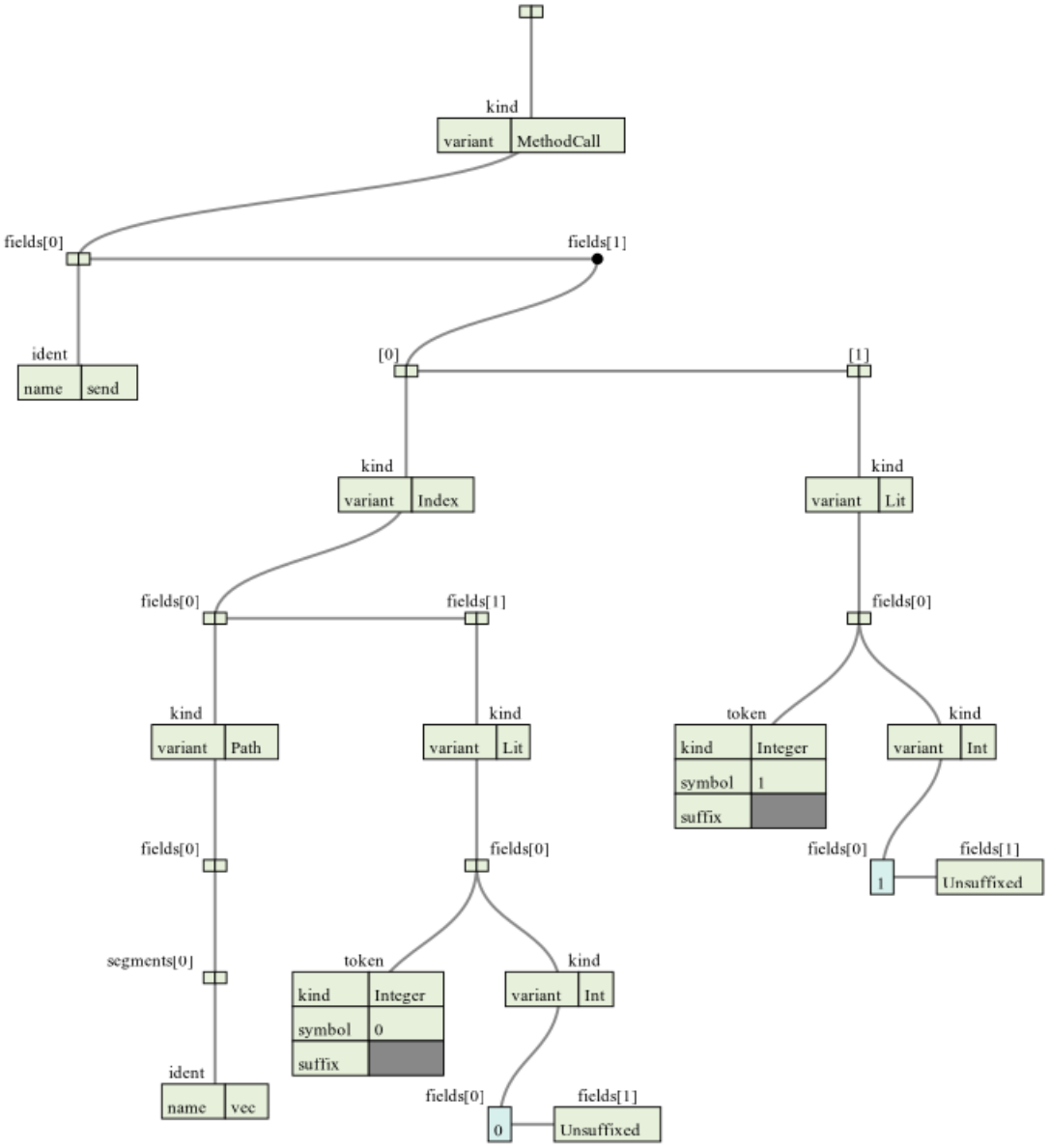

Fig. 4.13: Representación visual de AST sobre indexado de vector de SyncSender

4.3.8. Tuplas de canales

Tal como mencionamos anteriormente, la mayoría de las funciones de creación de canales retornan una tupla de 2 elementos, donde el primero es un Sender o SyncSender y el segundo es un Receiver. Para los casos en los que no se realiza un Tuple Unpack sino que se guarda la tupla completa en una variable, nosotros almacenamos la misma con tipo Tupla, junto con el tipo y la librería de los elementos que contiene. El acceso a los elementos de la tupla también es tenido en cuenta a la hora de contabilizar llamados a funciones sobre canales o asignaciones de variables. Por ejemplo: para los ExprKind de tipo MethodCall, si el primer argumento de la lista se corresponde con un ExprKind de tipo Field y este último contiene un ExprKind de tipo Path, el cual se corresponde a una variable de tipo Tupla en nuestro diccionario, entonces definimos en dicho diccionario una variable fresca

y contamos la ocurrencia del llamado a la función. En el Listing 4.13 hay un ejemplo de una creación de canal, su almacenamiento en una tupla y el aliasing del primer elemento de la tupla en otra variable. A su vez, en la Figura 4.14, se puede ver el fragmento de AST correspondiente al acceso al primer elemento de la tupla.

```
fn tuple_sync_sender_mpsc(){
    let channel = mpsc::sync_channel();
    let sync_sender = channel.0;
}
```

Listing 4.13: Ejemplo de canal como tupla

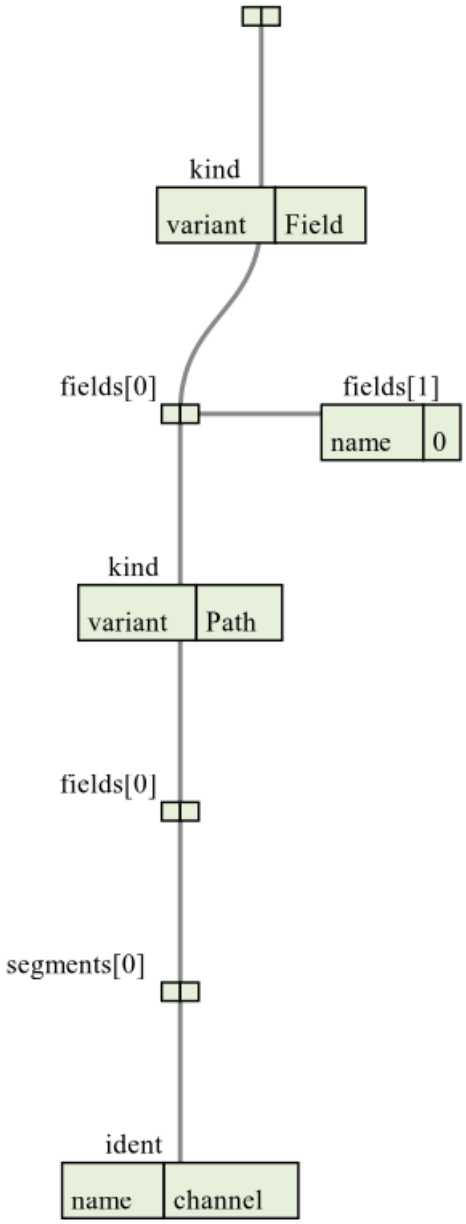

Fig. 4.14: Representación visual de AST sobre acceso a elemento de tupla

4.3.9. Argumentos de Funciones

En Rust, todas las funciones tienen que tener sus argumentos anotados con sus respectivos tipos. Gracias a esto, podemos contabilizar las funciones que se utilizan sobre argumentos que representan canales. A nivel AST, cuando visitamos un ItemKind de tipo Fn, analizamos su FnSig, el cual representa el *signature* de la función. Dentro de este elemento, tenemos un FnDecl (la declaración de la función propiamente dicha), el cual está compuesto de una lista de inputs y un output. La lista de inputs es un Vector de Param, los cuales contienen un elemento de tipo Ty, el cual representa al tipo de cada uno. Si este tipo corresponde con un Sender/SyncSender o Receiver, entonces agregamos una nueva variable a nuestro diccionario con el tipo del argumento para así poder contabilizar los llamados del cuerpo de la función. En el Listing 4.14 podemos ver un ejemplo de una función que recibe por parámetro un Sender para enviar un mensaje sobre él y en la Figura 4.15 tenemos el fragmento de AST correspondiente a la declaración de la función, donde se puede observar el tipo de su argumento.

```
fn params_crossbeam(sender: crossbeam_channel::Sender<i32>){
    sender.send(10);
}
```

Listing 4.14: Ejemplo de función con argumento de tipo Sender

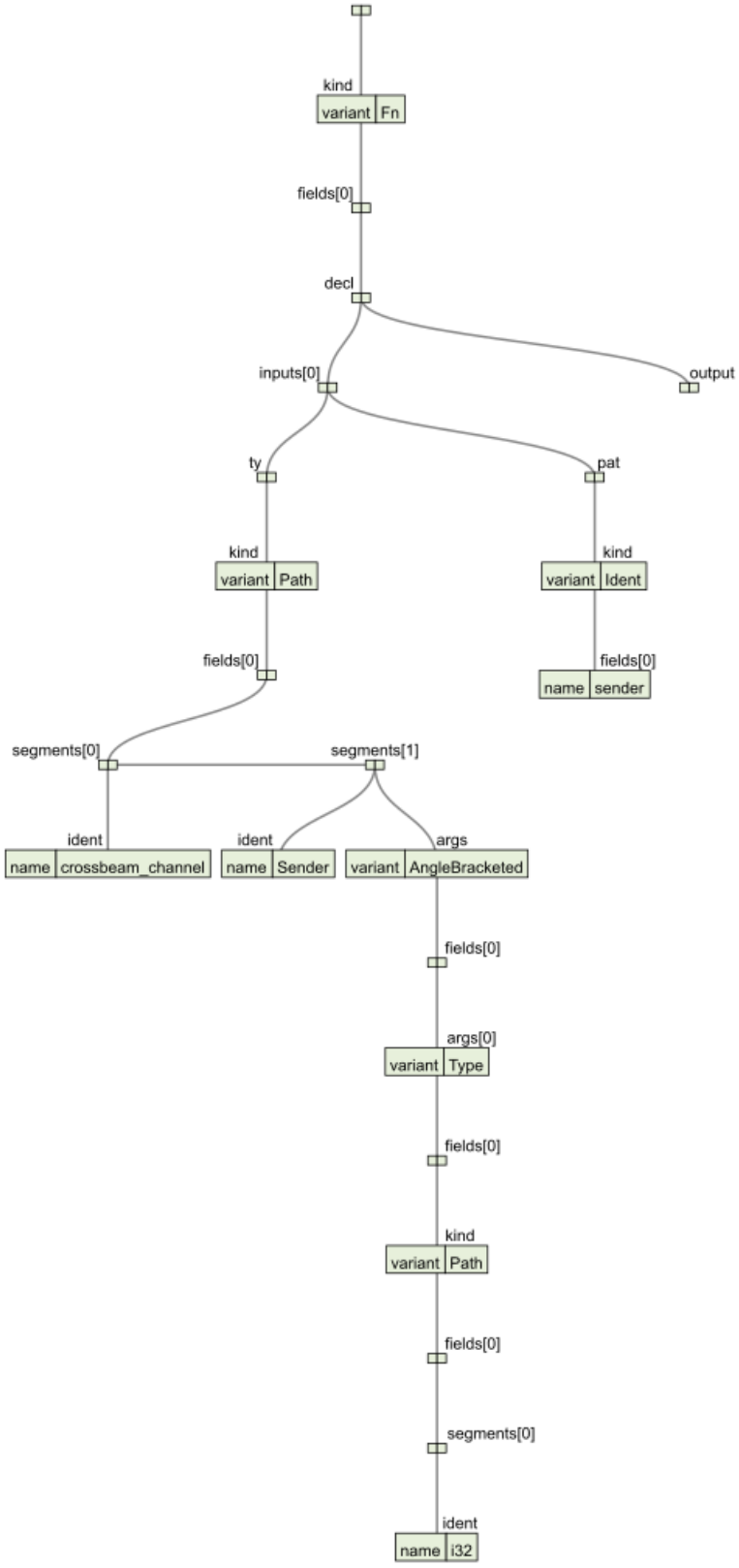

Fig. 4.15: Representación visual de AST sobre función con argumento de tipo Sender

5. EXPERIMENTACIÓN

En esta sección discutiremos los resultados del análisis cuantitativo sobre el uso de funciones de concurrencia, particularmente envío de mensajes sobre canales, en el lenguaje Rust, presentando los mismos mediante tablas numéricas y box plots.

5.1. Frecuencia de operaciones sobre canales en Rust

Feature	Proyectos	Proporción
channel	269	30.06 %
send	193	21.56 %
recv	193	21.56 %
select	5	0.56 %
iter	24	2.68 %
drop	30	3.51 %
sender clone	110	12.29 %
receiver clone	17	1.90 %

Tab. 5.1: Proyectos que utilizan funciones sobre canales

Del total de 900 proyectos, excluimos 5, los cuales o bien no poseen código Rust o bien contienen el código de las librerías de canales bajo análisis. La tabla 5.1 resume los resultados encontrados con respecto a la cantidad de operaciones sobre canales en los 895 proyectos analizados. En principio, podemos observar que sólo 269 (30 %) de los 895 proyectos crean canales de comunicación. Esto contrasta bastante con los resultados del análisis realizado en Go, donde ese valor asciende al 76 % en 865 proyectos. Con respecto a las funciones de send y recv, las mismas se utilizan en más del 21 % de los casos, mientras que un poco más atrás se ubica el clone de senders, con poco más del 12 %. La función menos utilizada es la de select, la cual apenas alcanza el 0.5 %, sin embargo es necesario recordar que en este análisis no se está teniendo en cuenta el uso de la macro *select!*. Para tener una noción sobre el uso de dicha macro, utilizamos el comando *grep* para encontrar aquellos proyectos que la incluyen, y observamos coincidencias en 60 de estos, con 417 ocurrencias en total. Esto sugiere que es mucho más común el uso de la macro sobre la variante dinámica, pero también hay que tener en cuenta que dentro de este número seguramente se estén contabilizando usos que correspondan a otras librerías (como *tokio* y *futures*, según notamos al realizar un análisis manual) y podría ocurrir también que algunas de estas apariciones formen parte de comentarios en el código. Continuando con el análisis de las demás funciones, se puede observar que existen muy pocos usos de clone de receivers en comparación con los de senders, lo cual posiblemente esté influenciado por el hecho de que la librería `MPSC` no ofrece este feature y de acuerdo a nuestros resultados, dicha librería es la que más fuertemente se utiliza de las 3 analizadas. Para el caso de los drop, el bajo uso puede deberse al hecho de que el esquema de memoria que maneja Rust ya se encarga implícitamente de llamar al destructor cuando una variable deja de pertenecer a su scope.

En el resto de esta sección, nos enfocaremos en los 269 proyectos que contienen al menos

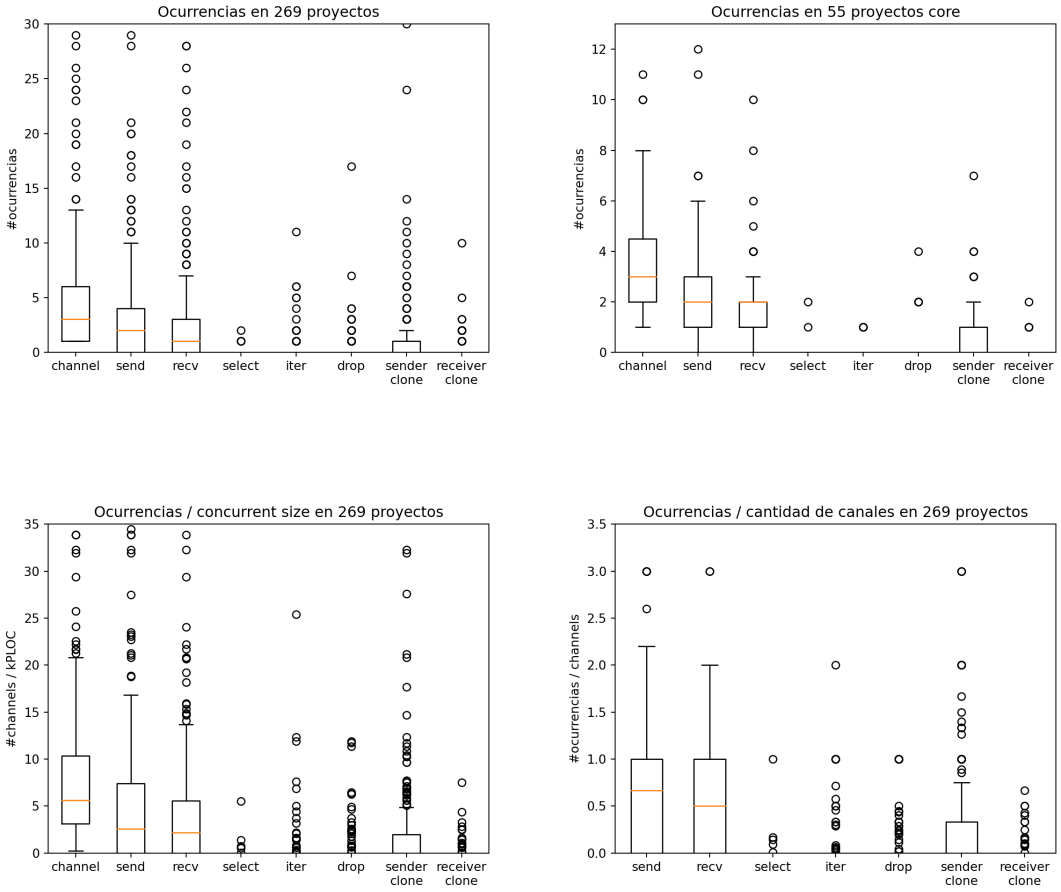

Fig. 5.1: Box plots para frecuencia de operaciones sobre canales en Rust

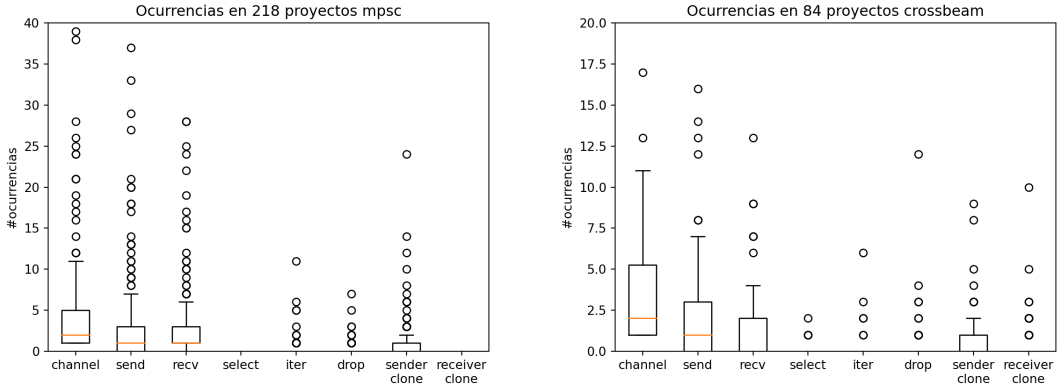

Fig. 5.2: Box plots para frecuencia de operaciones en `MPSC` y `Crossbeam`

una función de creación de canales. Presentamos nuestras mediciones tanto en términos absolutos como relativos, esta última con respecto al concurrent size de los proyectos y a la cantidad de ocurrencias de creación de canales.

Feature	mean	std	min	25 %	50 %	75 %	max
channel	7.33	19.07	1	1	3	6	206
send	4.87	13.28	0	0	2	4	130
recv	4.88	22.42	0	0	1	3	334
select	0.02	0.17	0	0	0	0	2
iter	0.23	1.03	0	0	0	0	11
drop	0.47	3.57	0	0	0	0	55
sender clone	1.66	7.67	0	0	0	1	117
receiver clone	0.15	0.79	0	0	0	0	10

Tab. 5.2: Ocurrencia absoluta en 269 proyectos

5.1.1. Mediciones absolutas

	mean	std	min	25 %	50 %	75 %	max
branches	1.67	1.21	1	1	1	2.5	4

Tab. 5.3: Cantidad de branches por select

La primera imagen de la figura 5.1 (arriba a la izquierda) y la tabla 5.2 muestran el promedio, desvío estándar, mínimo, máximo y los cuartiles sobre la cantidad de ocurrencias de funciones sobre canales en los 269 proyectos que contienen al menos una creación de canal. Es importante mencionar que en los box plots de esta sección se removieron *outliers* para una mejor visualización de los resultados. En promedio, los proyectos analizados contienen 7.33 ocurrencias de creaciones de canales, con una mediana de 3. La cantidad de ocurrencias promedio de send es de 4.87, con una mediana de 2, mientras que el promedio de recv es de 4.88, con una mediana de 1. Los clone de senders son los terceros más utilizados con un promedio de 1.66 y una mediana de 0. Por su parte, se puede apreciar que select, iter, drop y el clone de receivers no se utilizan con demasiada frecuencia. La tabla 5.3 analiza el tamaño de los select en términos de la cantidad de branches. Acá se puede observar que los select tienen en promedio entre 1 y 2 branches, siendo 4 la cantidad máxima observada.

Los 3 proyectos con mayor cantidad de creaciones de canales son tikv/tikv (concurrent size 49.76 kPLOC, ratio = 20.54 %), solana-labs/solana (concurrent size 63.38 kPLOC, ratio = 29.22 %) y apache/incubator-teaclave-sgx-sdk (concurrent size = 1.20 kPLOC, ratio = 1.66 %). El proyecto tikv, el cual corresponde a una base de datos transaccional distribuida de clave-valor, posee 206 creaciones de canales, 128 send, 334 recv, 4 drop, 2 iter y 117 clone de senders y 1 clone de receivers. Este proyecto utiliza tanto `MPSC` como `Crossbeam`, perteneciendo la gran mayoría de los canales a la primera librería. Los canales están repartidos en diferentes archivos del proyecto y se detectaron algunos usos interesantes de los mismos, tales como canales de canales (1 ocurrencia) o creaciones de canales en ciclos (4 ocurrencias). Además de ser el proyecto con mayor cantidad de creaciones de canales, es el que más usos de recv posee y el segundo proyecto con mayor cantidad de

send. El proyecto solana, el cual corresponde a un blockchain para marketplaces o apps descentralizadas rápidas, seguras y escalables, posee 153 creaciones de canales, 59 send, 34 recv, 17 drop, 4 iter, 11 clones de senders y 1 select. Casi un 55 % de los canales utilizados corresponde a `Crossbeam`, mientras que el resto a `MPSC`. También se detectaron usos interesantes tales como arrays de canales (3 ocurrencias) y canales de canales (3 ocurrencias). Al igual que en el anterior proyecto, los canales están repartidos en varios archivos, y también existen algunos en los que se manejan a la vez canales de ambas librerías. Con respecto al Select detectado, en el mismo se utilizan sólo 2 branches, ambos con un recv en sus guardas. El proyecto incubator-teaclave-sgx-sdk, utilizado para escribir aplicaciones Intel SGX en lenguaje Rust, posee 123 creaciones de canales (todas de `MPSC`), 130 send, 114 recv, 55 drop, 6 iter y 14 clone de senders. Aproximadamente, el 70 % de los canales acotados detectados tiene una capacidad de 0, el 25 % es de capacidad 1, un 4 % es de capacidad 3 y un 1 % es de capacidad 10000. Sin embargo, es importante mencionar que la gran mayoría de las ocurrencias de funciones de canales se dieron en un solo archivo del proyecto, el cual es un test unitario sobre la librería `MPSC`. Los restantes se dieron en otro archivo, el cual es un test unitario sobre threads. Este proyecto es a su vez el que más ocurrencias de send posee y el segundo con mayor cantidad de recv.

En total, se detectaron 218 proyectos con canales de `MPSC`, 84 de `Crossbeam` y sólo 4 de `Flume`. En la figura 5.2 se pueden ver los resultados del análisis para las primeras dos librerías. En el caso de `MPSC`, los proyectos tienen en promedio 6.27 creaciones de canales, con una mediana de 2. La cantidad de ocurrencias promedio de send es de 4.74, con una mediana de 1, y la de recv es de 5.11, también con una mediana de 1. Para el caso de iter, drop y el clone de senders, se detectaron en promedio 0.22, 0.44 y 1.54 ocurrencias de cada una respectivamente, todas con una mediana de 0. Dado que `MPSC` no soporta Select ni clone de receivers, no hay ocurrencias de los mismos. Con respecto a `Crossbeam`, los proyectos tienen en promedio 6.89 creaciones de canales, 3.13 ocurrencias de send y 2.27 ocurrencias de recv, con medianas de 2, 1 y 0 respectivamente. Para el caso de select, iter y drop se detectaron en promedio 0.07, 0.18 y 0.37 ocurrencias de cada una respectivamente, todos con una mediana de 0. A su vez, los proyectos analizados contienen 1.20 ocurrencias de clone de senders y 0.45 ocurrencias de clone de receivers, también con una mediana de 0. Por último, si bien la cantidad de proyectos que utilizan `Flume` detectada es muy baja como para ser estadísticamente relevante, mencionamos igualmente que en los mismos detectamos en promedio 6.25 ocurrencias de creaciones de canales, 3.75 de send, 2.25 de recv, 2 clone de senders y 0.50 clone de receivers.

Para comparar el uso de funciones de canales en términos absolutos sobre proyectos de tamaño concurrente similares se seleccionaron aquellos cuyo concurrent size cae dentro del 10 % del concurrent size medio de los 269 proyectos. La mediana del concurrent size de nuestra muestra es de 0.81 kPLOC, mientras que los *proyectos core* consisten en aquellos cuyo tamaño oscila entre 0.62 kPLOC y 1.16 kPLOC. Esto nos da un total de 55 proyectos core. La segunda imagen de la figura 5.1 (arriba a la derecha) y la tabla 5.4 resumen nuestros resultados. Como se puede apreciar, estos proyectos contienen en promedio 3.80 ocurrencias de creaciones de canales, 2.44 ocurrencias de send y 1.89 ocurrencias de recv, con medianas de 3, 2 y 2 respectivamente.

Dentro de estos 55 proyectos core, aquellos con mayor cantidad de creaciones de canales son el rust-lang/book (concurrent size 1.09 kPLOC, ratio = 10.22 %) y el jmacdonald/amp (concurrent size 0.82 kPLOC, ratio = 9.50 %). El primero de estos contiene el código fuente del libro *The Rust Programming Language* y cuenta con archivos de ejemplo que contie-

nen 21 creaciones de canales (todas de `MPSC`), 6 send, 2 recv y 1 clone de sender. Por otro lado, el proyecto amp, el cual es un editor de texto en consola basado en vim, posee 11 creaciones de canales (todas de `MPSC`), 1 send, 1 recv y 2 clone de sender. El proyecto con mayor cantidad de send es el hatoo/oha (concurrent size 0.79 kPLOC, ratio = 53.46 %). El mismo es un pequeño programa que envía carga a una aplicación web y muestra la información en una interfaz de texto por consola en tiempo real. Cuenta con 8 creaciones de canales (todas de `Flume`), 12 send, 8 recv, 4 clones de sender y 2 clone de receivers. A su vez, el proyecto con mayor cantidad de recv es el actix/actix-net (concurrent size 0.65 kPLOC, ratio = 5.69 %). Dicho proyecto es una colección de librerías de bajo nivel para servicios de red y posee 10 creaciones de canales (todas de `MPSC`), 7 send y 10 recv. En todos estos casos, los usos de canales detectados fueron básicos.

Feature	mean	std	min	25 %	50 %	75 %	max
channel	3.8	3.49	1	2	3	5	21
send	2.44	2.64	0	1	2	3	12
recv	1.89	1.98	0	1	2	2	10
select	0.05	0.30	0	0	0	0	2
iter	0. 07	0.26	0	0	0	0	1
drop	0.22	0.74	0	0	0	0	4
sender clone	0.85	1.39	0	0	0	1	7
receiver clone	0.09	0.35	0	0	0	0	2

Tab. 5.4: Ocurrencia absoluta en 55 proyectos core

5.1.2. Mediciones relativas al concurrent size

El primer tipo de mediciones relativas son con respecto al concurrent size de los proyectos. Para cada proyecto P, se divide la cantidad de ocurrencias de cada feature por $|P|$. La tercera imagen de la figura 5.1 (abajo a la izquierda) y la tabla 5.5 resumen los datos encontrados. En promedio, se utilizan 9.21 funciones de creación de canales por cada 1000 líneas de código concurrente, con una mediana de 5.60. A su vez, la cantidad relativa promedio de ocurrencias de send y recv es de 7.10 y 6.29.

Feature	mean	std	min	25 %	50 %	75 %	max
channel	9.21	11.89	0.23	3.12	5.60	10.37	102.67
send	7.10	14.04	0	0	2.59	7.55	108.51
recv	6.29	14.74	0	0	2.16	5.62	142.86
select	0.03	0.35	0	0	0	0	5.51
iter	0.60	4.52	0	0	0	0	66.67
drop	0.53	3.17	0	0	0	0	45.91
sender clone	2.18	5.97	0	0	0	2.03	66.67
receiver clone	0.12	0.65	0	0	0	0	7.49

Tab. 5.5: Ocurrencias relativas con respecto al concurrent size en 269 proyectos

El proyecto con mayor cantidad de creación de canales por concurrent size es incubator-teaclave-sgx-sdk, con un valor de 102.67. Dicho proyecto es también el que posee la mayor cantidad de send por concurrent size, (con un valor de 108.51) y el tercero con mayor

cantidad de recv por concurrent size (con un valor de 95.16). El proyecto con mayor cantidad de recv por concurrent size es daboross/fern (herramienta de logging simple y eficiente), con un valor de 142.86. El mismo también es el segundo con mayor cantidad de creaciones de canales por concurrent size (con un valor de 71.43). Sin embargo, tenemos que mencionar que el motivo por el cual posee valores tan altos es porque el concurrent size de este proyecto es solo de 14, dado que sólo se utilizan canales en un solo archivo, el cual consta únicamente de un pequeño test.

5.1.3. Mediciones relativas a la cantidad de canales

El segundo tipo de mediciones relativas son con respecto a la cantidad de funciones de creación de canales de los proyectos. Esta medida brinda un aproximado de la cantidad de funciones que se invocan en cada canal. La cuarta imagen de la figura 5.1 (abajo a la derecha) y la tabla 5.6 resumen los datos encontrados. En promedio, la cantidad de ocurrencias por canal de cada una de las funciones analizadas oscila entre 0 y 1, lo cual sugiere que los canales se utilizan para protocolos de sincronización muy simples.

Feature	mean	std	min	25 %	50 %	75 %	max
send	0.75	0.87	0	0	0.67	1	6
recv	0.57	0.54	0	0	0.50	1	3
select	0.01	0.06	0	0	0	0	1
iter	0.05	0.21	0	0	0	0	2
drop	0.06	0.21	0	0	0	0	1
sender clone	0.29	0.56	0	0	0	0.37	4
receiver clone	0.02	0.08	0	0	0	0	0.67

Tab. 5.6: Ocurrencias relativas con respecto a la cantidad de canales en 269 proyectos

El proyecto con mayor cantidad de send por canal es el mit-pdos/noria, con un valor de 6. El mismo es un sistema de streaming data-flow diseñado para trabajar como backend de almacenamiento rápido para aplicaciones web de lectura intensiva. Por otro lado, el proyecto con mayor cantidad de recv por canal es el cogciprocate/ocl, con un valor de 3. Este último consta de bindings e interfaces de OpenCL puro para Rust. En ambos casos, los proyectos sólo tuvieron una creación de canales, por lo que ése es el motivo por el cual la medición arroja un valor tan alto en comparación al resto.

Un posible motivo por el cual las ocurrencias de cada feature en relación a la cantidad de canales arrojan un valor tan bajo puede ser por limitaciones propias de este análisis estático de código. Si estos features se wrapean en otras funciones y las mismas son llamadas desde varios puntos del código, para nuestra herramienta se contabilizan sólo como una ocurrencia, pero en tiempo de ejecución está claro que los features se van a utilizar más veces.

5.2. Proporción de concurrencia

En esta sección, analizamos la proporción de proyectos Rust relacionados con la concurrencia. Para esto, consideramos 2 medidas distintas: la cantidad de líneas físicas de código y la cantidad de archivos. La primera imagen de la figura 5.3 y la tabla 5.7 resumen los resultados para los 269 proyectos que contienen al menos una función de creación

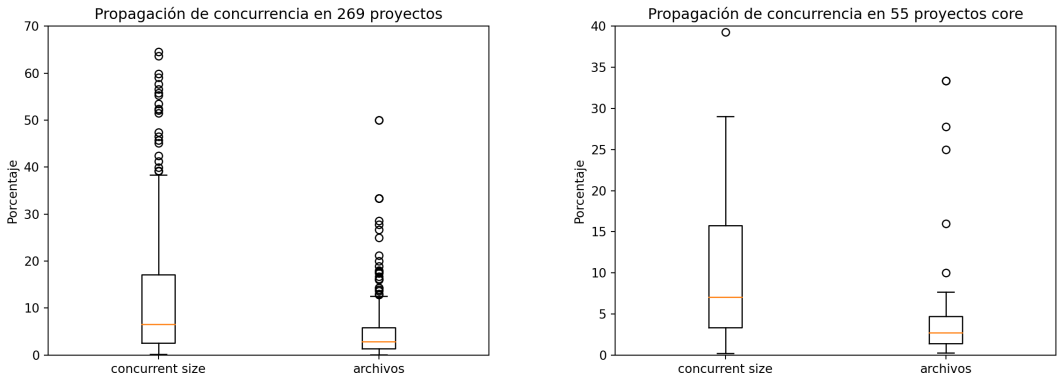

Fig. 5.3: Box plots para proporción de concurrencia en proyectos Rust

de canales. La primera fila de la tabla muestra el radio del concurrent size con respecto a la cantidad total de líneas físicas de código en los proyectos. La tabla muestra que, en promedio, 13.30 % de las líneas de códigos son sobre concurrencia, con una mediana de 6.57 %. La segunda fila de la tabla muestra el radio del número de archivos con features de concurrencia con respecto al total de archivos. En promedio, 5.32 % de los archivos de un proyecto contienen al menos un feature de concurrencia, con una mediana de 2.86 %.

La segunda imagen de la figura 5.3 y la tabla 5.8 dan los resultados del mismo análisis para los 55 proyectos core descriptos anteriormente.

Medida	mean	std	min	25 %	50 %	75 %	max
concurrent size	13.30	16.74	0.12	2.58	6.57	17.13	96.69
archivos	5.32	7.24	0.05	1.40	2.86	5.88	50

Tab. 5.7: Proporción de concurrencia en 269 proyectos

Medida	mean	std	min	25 %	50 %	75 %	max
concurrent size	14.71	20.72	0.20	3.33	7.06	16.87	96.69
archivos	5.90	9.68	0.27	1.38	2.73	4.76	50

Tab. 5.8: Proporción de concurrencia en 55 proyectos core

A grandes rasgos, podemos observar que si bien existen menos archivos con features de concurrencia que archivos secuenciales, los primeros suelen tener un mayor tamaño en comparación. Esto se puede concluir a partir del hecho de que la proporción de concurrent size es mayor con respecto a la de cantidad de archivos con features de concurrencia.

5.3. Frecuencia de uso de canales infinitos vs. acotados

Las librerías de Rust en general ofrecen 2 tipos de canales: con capacidad infinita y acotados. En los primeros, todas las operaciones de send son no bloqueantes, mientras que en los segundos, esto solo sucede hasta alcanzar la capacidad máxima del canal. Una vez ocurrido esto, los canales acotados tienen un comportamiento sincrónico, es decir, los

send son bloqueantes. En este trabajo analizamos la frecuencia de uso de ambos tipos de canales. Para el caso de canales acotados, investigamos también la frecuencia con la que la capacidad puede ser obtenida estáticamente. Dentro de este grupo, también analizamos como un caso especial los canales de capacidad 0, es decir, aquellos que son completamente sincrónicos.

En la tabla 5.9 podemos ver los resultados obtenidos. Los proyectos analizados contienen aproximadamente 1971 canales. De estos, el 80 % corresponde a canales de capacidad infinita. Más del 15 % corresponde a canales acotados cuya capacidad fue posible determinar de manera estática, mientras que poco más del 4 % corresponde a canales acotados cuya capacidad es desconocida. Más del 5 % corresponde a canales de capacidad 0 conocida, mientras que más el 90 % corresponde a canales con capacidad estrictamente mayor a 0 conocida.

Tipo	**Ocurrencias**	**Proporción**
Todos los canales	1971	100 %
Canales con cotas conocidas	308	15.63 %
Canales síncronos (conocidos)	110	5.58 %
Canales asíncronos (conocidos)	1775	90.06 %
Canales con cotas desconocidas	86	4.36 %
Canales de capacidad infinita	1577	80.01 %

Tab. 5.9: Canales de comunicación en 269 proyectos

La tabla 5.10 brinda los resultados sobre el tamaño de los canales acotados asincrónicos de capacidad conocida. La gran mayoría de estos tiene capacidad 1 y puede observarse la presencia de un outlier de 100000 que afecta el cálculo del promedio y del desvío estándar.

	mean	std	min	25 %	50 %	75 %	max
Capacidad	2837.72	14868.45	1	1	1	100	100000

Tab. 5.10: Tamaños conocidos de canales asincrónicos

Creemos que el fuerte uso que se da en la práctica de los canales infinitos puede estar influenciado por el hecho de que son los canales que ofrece por default Rust. Trazando un paralelismo con el trabajo de Go, en éste se puede ver como la mayoría de los canales utilizados en la práctica son los sincrónicos, pero estos también son los default que ofrece dicho lenguaje, lo cual refuerza nuestra teoría.

5.4. Topologías de concurrencia

En esta sección, investigamos si los programas analizados contienen o no topologías de concurrencia complejas. Para esto, analizamos la ocurrencia de de creaciones de threads y canales en ciclos así como el almacenamiento de canales en listas, vectores, tuplas u otros canales.

5.4.1. Creación de Threads

La primera parte de la tabla 5.11 muestra la frecuencia de distintos patrones de thread spawns en los 895 proyectos analizados. De acuerdo a la tabla, el 39.44 % de los proyectos

Feature	Proyectos	Proporción
Spawn	353	39.44 %
Spawn en cualquier ciclo	121	13.52 %
Spawn en ciclos con cota conocida	36	4.02 %
Spawn en ciclos con cota desconocida	102	11.40 %
Canales en cualquier ciclo	32	3.58 %
Canales en ciclos con cota conocida	4	0.45 %
Canales en ciclos con cota desconocida	30	3.35 %
Aliasing de canales en ciclos	0	0 %
Canales en arrays	1	0.11 %
Canales en vectores	3	0.34 %
Canales en tuplas	1	0.11 %
Canales de canales	2	0.22 %

Tab. 5.11: Frecuencia de ocurrencia de features en todos los proyectos

utilizan spawn de threads y el 13.52 % lo hace dentro de algún ciclo. Distinguimos el caso de ciclos con cota conocida, es decir, for loops donde la cantidad de iteraciones se conoce estáticamente, y aquellos de cota desconocida, es decir, while, loop y for loop donde no se puede conocer la cantidad de iteraciones de manera estática (es decir, no están definidas como literales en la guarda). Solo un 4.02 % de los proyectos crean threads en ciclos con cotas conocidas, mientras que el 11.40 % lo hace sin cotas conocidas. La tabla 5.12 muestra información sobre las cotas conocidas para aquellos ciclos que contienen spawn, mientras que la primera parte de la tabla 5.13 muestra las ocurrencias relativas de estos features en proyectos que contienen al menos una ocurrencia de dichos features.

	mean	std	min	25 %	50 %	75 %	max
Cota	128.13	676.68	1	5	9	10.75	5000

Tab. 5.12: Cotas conocidas para ciclos con spawn

5.4.2. Creación de Canales

La segunda parte de la tabla 5.11 muestra la proporción de proyectos en las que se crean canales dentro de ciclos. Como se puede apreciar, la cantidad de proyectos que utilizan este patrón (3.58 %) es menor a la de aquellos que crean threads dentro de ciclos (13.52 %). Sólo en el 0.45 % de los proyectos se pudo determinar una cota para este tipo de ciclos de manera estática, mientras que en el 3.35 % no. La segunda parte de la tabla 5.13 muestra la cantidad relativa de estos patrones en proyectos que contienen al menos una ocurrencia de dicho patrón. Es interesante mencionar que no se encontraron ocurrencias de aliasing de canales en ciclos en ninguno de los proyectos analizados.

5.4.3. Almacenamiento de Canales

La última parte de la tabla 5.11 muestra la cantidad de proyectos para los cuales se utilizan estructuras para almacenar canales. De acuerdo a los resultados obtenidos, menos del 1 % de los proyectos utilizan estos patrones.

Patrones	mean	std	min	25 %	50 %	75 %	max
Spawn	8.93	11.61	0.22	2.73	4.88	10	89.55
Spawn en cualquier ciclo	4.24	7.26	0.04	0.67	1.63	4.57	55.56
Spawn en ciclos con cota conocida	2.44	3.46	0.03	0.59	0.92	2.62	14.92
Spawn en ciclos con cota desconocida	4.17	7.41	0.03	0.60	1.52	4.25	55.56
Canales en cualquier ciclo	1.01	1.74	0.04	0.20	0.44	0.98	8.35
Canales en ciclos con cota desconocida	1.03	1.79	0.04	0.19	0.44	0.91	8.35

Tab. 5.13: Ocurrencias relativas con respecto al concurrent size

5.5. Limitaciones del estudio

Tal como se mencionó con anterioridad, en este trabajo sólo se analizan 3 librerías de canales. En Rust existen varias otras librerías, tales como *chan* o *futures*, por lo que no podemos contabilizar el uso total de canales en el lenguaje.

Para el caso de los Select, sólo se tuvo en cuenta la variante dinámica de la misma, dejando de lado por simplicidad el uso de la macro select!. Esto repercute en el hecho de que seguramente se estén contando menos ocurrencias de las que realmente existen.

Para el caso de la iteración sobre canales, solo se tuvo en cuento los llamados explícitos a la función iter. En algunas librerías es posible directamente iterar sobre el Receiver sin necesidad de llamar a ninguna función, pero estos casos no son tenidos en cuenta en el análisis por simplicidad. El proyecto ripgrep utiliza justamente este patrón.

Durante la ejecución del comando experimental de rustc para obtener el AST de cada archivo Rust, existieron varios archivos los cuales arrojaron errores durante el parseo. En total fueron 520 archivos en 90 proyectos, los cuales terminamos analizando a mano. Sin embargo, de todos esos, solo 1 contenía uso de canales. Los errores más frecuentes fueron:

- Archivos como templates.
- Archivos con errores sintácticos.
- Archivos vacíos.
- Macros que no se encontraban en el propio archivo.
- Nombres de archivos con puntos.

Es muy probable que la cantidad de funciones utilizadas se estén aproximando por debajo del número real. Si un programador wrapea las funciones de send o recv en funciones propias y luego las llama desde otros puntos del código, estas ocurrencias no se contabilizan. Si se definen Structs que posean canales como propiedades y luego en un Impl se utilizan, tampoco se tienen en cuenta los llamados a funciones sobre las mismas.

Otro caso que notamos en algunos proyectos fue el uso de Watchers, los cuales pertenecen a la librería Notify, que se utiliza para observar cambios en el file system. Estos Watchers reciben como parámetro un Sender y cada vez que detectan una modificación en el directorio del file system especificado, envían una notificación a través del mismo. Dichas ocurrencias de send, al ejecutarse desde la librería, no son contabilizadas.

Como se mencionó con anterioridad, la librería `MPSC` no brinda la posibilidad de realizar clone de Receivers, sin embargo, notamos que en la práctica para solventar esta limitación, varios proyectos utilizar un struct de la librería estándar denominado Arc. El mismo es

un puntero a una posición de memoria, el cual sí admite la función de clone, lo cual posibilita el ownership compartido de valores (por ejemplo, Receivers). Los usos de funciones de canales a través de Arc no son tenidos en cuenta en este análisis.

Con respecto a las capacidades de los canales, solamente tenemos en cuenta aquellas que son definidas estáticamente al momento de invocar la función de creación de canal.

6. CONCLUSIONES

Luego de haber analizado 895 proyectos Rust públicos de GitHub, encontramos que menos de un tercio de los mismos (30 %) utilizan canales. Las topologías de concurrencia son en su mayoría simples, puesto que muy pocos proyectos crean canales en ciclos o los almacenan en estructuras de datos tales como colecciones. A su vez, le cantidad de usos de las funciones de send y recv por canal es bastante baja (alrededor de 1 en promedio). De la misma manera, la proporción de código de cada proyecto dedicada a features de concurrencia es también baja (13 % en promedio). Con respecto al tipo de canales que se utilizan, detectamos que la gran mayoría son asincrónicos (90 %), destacándose en este grupo aquellos con capacidad infinita (80 %), los cuales son los canales que brinda por defecto la librería estándar de Rust. Vimos que de las tres librerías analizadas, `MPSC` (de la librería estándar), a pesar de sus limitaciones, es la más utilizada (81 % de los proyectos) con una amplia diferencia sobre las demás. `Crossbeam` se ubica en segundo lugar con un 31 % de los proyectos y muy por detrás aparece `Flume`, con solo el 1 % de los mismos.

Como trabajo a futuro sería interesante repetir este análisis incluyendo los usos de canales en Structs/Impls, los cuales quedaron fuera del scope de este trabajo pero que observamos un uso recurrente de los mismos en varios de los proyectos revisados de forma manual. Además sería de utilidad expandir el alcance del trabajo, incorporando otras librerías que también se utilizan en la práctica, o intentando subsanar alguna otra de las limitaciones mencionadas anteriormente.

Bibliografía

[1] N. Dilley and J. Lange, "An empirical study of messaging passing concurrency in go projects," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 377–387, IEEE, 2019.

[2] P. Anderson, "The use and limitations of static-analysis tools to improve software quality," in *CrossTalk: The Journal of Defense Software Engineering, 21(6):18–21*, 2008.

[3] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen, "Session types for rust," in *Proceedings of the 11th acm sigplan workshop on generic programming*, pp. 13–22, 2015.

[4] R. Chen and S. Balzer, "Ferrite: A judgmental embedding of session types in rust," *arXiv preprint arXiv:2009.13619*, 2020.

[5] N. Lagaillardie, R. Neykova, and N. Yoshida, "Implementing multiparty session types in rust," in *International Conference on Coordination Languages and Models*, pp. 127–136, Springer, 2020.

[6] W. Kokke, "Rusty variation: deadlock-free sessions with failure in rust," *arXiv preprint arXiv:1909.05970*, 2019.

[7] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, (New York, NY, USA), p. 103–104, Association for Computing Machinery, 2014.

[8] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 763–779, 2020.

[9] Z. Yu, L. Song, and Y. Zhang, "Fearless concurrency? understanding concurrent programming safety in real-world rust software," *arXiv preprint arXiv:1902.01906*, 2019.

[10] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[11] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," in *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, (New York, NY, USA), p. 48–64, Association for Computing Machinery, 1998.

[12] A. Turon, "Fearless concurrency with rust," 2015.

[13] "Rust book, threads." `https://doc.rust-lang.org/book/ch16-01-threads.html`. Accedida el 05/09/2020.

[14] "Oracle, multithreading models." https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqk/index.html#ch2mt-41. Accedida el 05/09/2020.

[15] "Rust book, message passing." https://doc.rust-lang.org/book/ch16-02-message-passing.html. Accedida el 05/09/2020.

[16] A. Gerrand, "Share memory by communicating," *The Go Blog*, 2010.

[17] "Mpsc documentation." https://doc.rust-lang.org/nightly/std/sync/mpsc/index.html. Accedida el 05/09/2020.

[18] "Crossbeam documentation." https://docs.rs/crossbeam-channel/0.4.3/crossbeam_channel/index.html. Accedida el 05/09/2020.

[19] "Flume documentation." https://docs.rs/flume/0.8.4/flume/index.html. Accedida el 05/09/2020.

[20] "Rust book, ownership." https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html. Accedida el 05/09/2020.

[21] "Go official documentation, channel of channels." https://golang.org/doc/effective_go.html#chan_of_chan. Accedida el 05/09/2020.

[22] "Count lines of code." https://github.com/AlDanial/cloc. Accedida el 10/08/2020.