UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

# ESTUDIO E IMPLEMENTACIÓN DE UN LENGUAJE DE AUTORIZACIÓN PARA BASES DE DATOS RELACIONALES

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

VICENTE REYES VALDIVIESO

PROFESOR GUÍA:
FEDERICO OLMEDO BERÓN

PROFESOR CO-GUÍA:
MATÍAS TORO IPINZA

MIEMBROS DE LA COMISIÓN:
LUIS MATEU BRULÉ
ÉRIC TANTER

SANTIAGO DE CHILE
2023

# Resumen

La popularidad de nuevas arquitecturas de software, y la disponibilidad de novedosas formas de controlar la seguridad de datos en una aplicación, motivan a estudiar el problema de la autorización: Qué usuarios pueden hacer qué acciones con qué datos de una aplicación. En este trabajo, se analizan opciones disponibles, se identifica un hueco para el que las soluciones disponibles son insuficientes, y se propone una nueva.

La solución consiste en un lenguaje de dominio específico para definir reglas de autorización. El lenguaje, llamado Pilpilang, permite asociar permisos con usuarios y recursos de la aplicación. Está diseñado para transformarse en código SQL, que se pueda usar para configurar una base de datos PostgreSQL, de modo que el motor de base de datos sea el encargado de aplicar las reglas cuando y donde corresponda. El lenguaje también se implementa en un compilador que da ciertas garantías de correctitud, como el tipado estático.

Para evaluar la eficacia del trabajo realizado, se traducen tres conjuntos de ejemplos de autorización establecidos a Pilpilang, y se discuten las diferencias entre las versiones traducidas y las originales. Los resultados de estas comparaciones son positivos sobre la aplicabilidad del nuevo lenguaje, aunque son mixtos sobre su valor agregado sobre la creación de SQL directamente. Sin embargo, se identifican funcionalidades extra que pueden dar este valor agregado, las cuales quedan como trabajo futuro.

*A mi gato*

# Agradecimientos

Deseo agradecer a todas las personas que de una forma u otra permitieron la realización de este trabajo de investigación, en especial a mi tío Matías por motivarme a seguir esta carrera, A mis padres, Daniel e Ignacia por infinito cariño, paciencia, comprensión y apoyo desde siempre. A mis familiares, presentes y pasados, quienes sin escatimar esfuerzo han sacrificado gran parte de su vida por mí y me han formado y educado, que la ilusión de su existencia ha sido verme convertida en persona de provecho, mil gracias.

De igual manera a mis profesores guía que me han brindado su apoyo durante todo el desarrollo de este proyecto. A todos mis maestros que a lo largo de mis estudios aportaron sus conocimientos invaluables, sugerencias, apoyo y sobre todo por su gran paciencia.

Y a mis amigos, cómo no.

# Tabla de Contenido

# Capítulo 1

# Introducción

Hoy en día, las aplicaciones trabajan con más datos que nunca, muchos de los cuales son datos sensibles, por lo que la seguridad es cada vez más relevante. Estas aplicaciones suelen permitir o restringir el acceso o cambios a sus datos dependiendo de quién las está usando, los permisos, roles, relaciones que tengan estos usuarios. Esta tarea consiste en dos partes:

1. Autentificación: Asegurarse de que los usuarios sean quienes dicen ser, por ejemplo, a través de una contraseña y luego una *cookie*.
2. Autorización: Sabiendo qué usuario está usando la aplicación, decidir a qué recursos tiene acceso, y qué cosas puede hacer con estos recursos.

La autentificación es un problema estándar, para el que existen múltiples librerías y soluciones que se pueden incluir y hacer funcionar rápidamente en una aplicación [15, 3], además de la tradicional solución del usuario y contraseña. Autentificar a usuarios suele ser ortogonal a la lógica de negocio de la aplicación, por lo que es inusual que estas librerías requieran mucha configuración.

Por otro lado, está la autorización. En todo servicio que tenga usuarios siempre habrá motivos para configurar autorización, motivos que serán distintos para cada aplicación, dependiendo de sus características. Por ejemplo, en un servicio como Twitter, los datos son públicos, pero si un usuario bloquea a otro, ese otro no podrá escribir respuestas a sus tweets. En otros servicios, como GitHub, algunos datos pueden ser privados, y modificarlos puede estar regido por otras reglas. Puede que haya que configurar organizaciones, que contengan usuarios y distribuyan permisos entre ellos.

## 1.1. Motivación

Se ha mencionado que en cada aplicación, la forma en que se maneja la autorización será distinta. Las funcionalidades o datos que se permita o prohíba usar dependerán del propósito de la aplicación y de su diseño en general. Por este motivo, no hay soluciones generales que

sirvan para todos, y el código de una aplicación[1] puede llegar a contener muchas líneas que describan la solución particular que esta usa.

La complejidad de los problemas de autorización se vuelve particularmente difícil de manejar en situaciones que surgen de arquitecturas de software modernas, como las de microservicios, que pueden utilizar varios lenguajes de programación distintos para ejecutar sus tareas, dentro de la misma compañía. Si las reglas de autorización están escritas en un lenguaje, pero el microservicio se quiere escribir otro, será necesario duplicar ese trabajo para que las reglas sean consistentes entre los distintos servicios. Por este motivo, recientemente ha habido un resurgimiento en la creación de soluciones para la autorización. Sin embargo, estas soluciones están diseñadas para integrarse al *back-end* de la aplicación, lo cual tiene problemas:

- El *back-end* de las aplicaciones es muchas veces la parte vulnerada por atacantes, ya que tiene comparativamente menos inversión en seguridad que otras partes, como el sistema operativo o la base de datos. Encargarse de la autorización en la BBDD es delegar los detalles a un proyecto que ya ha enfrentado estos problemas a lo largo del tiempo, por lo que es más probable que los haya solucionado.
- La autorización por software puede llevar a repetición de código. Autorizar en BBDD es buen diseño, ya que los datos son justamente lo que se debe proteger, por lo que las responsabilidades de protegerlos y proveerlos están intrínsicamente relacionadas.

## 1.2. Objetivos

A continuación, se exponen los objetivos que se pretende lograr en el presente trabajo.

### 1.2.1. Objetivo General

Se diseña Pilpilang, un DSL que adopta los patrones de un lenguaje de autorización moderno existente. Conjuntamente, se implementa un transpilador, es decir, un programa que lee código Pilpilang, y emite como resultado código de autorización SQL equivalente. El código SQL generado crea reglas suficientemente robustas como para que con él y una capa que provea autenticación y HTTP, se pueda servir públicamente la base de datos, sin que un usuario malicioso pueda acceder a datos o modificar datos que no debería.

### 1.2.2. Objetivos Específicos

El núcleo del trabajo es diseñar Pilpilang e implementar su transpilador, por lo que los objetivos específicos son los comunes al escribir un compilador. Sin embargo, se añaden las etapas sobre relacionar el código leído con la estructura de la base de datos dada.

[1] En general el *Back-end*

1. Diseñar el DSL, lo cual involucra estudiar las funcionalidades y patrones que son estado del arte en la definición de permisos, determinar su importancia y la viabilidad de implementarlos en un transpilador a SQL.
2. Crear un módulo que transforme el lenguaje a una representación interna: Una de las primeras capas en cualquier compilador es la de *parsing*, capa que lee el código fuente y lo transforma en una representación útil para el compilador.
3. Análogamente a la etapa anterior, se debe obtener la estructura de la base de datos (tablas, columnas y usuarios) y llevarla a una representación interna útil para el compilador.
4. Comprobar la factibilidad de las reglas definidas, relacionadas a la base de datos dada, para informar en caso de error. Es decir, si el código Pilpilang hace referencia a una columna que no existe en la base de datos, el proceso de compilación se debe interrumpir. Esto significa hacer un chequeo de que se usen los tipos correctos en las distintas reglas.
5. Recorrer las reglas definidas, emitiendo código SQL que las implemente.

## 1.3. Estado del arte

Hay una multitud de soluciones existentes para organizar las decisiones de autorización, ya que la seguridad informática es un tema que lleva siendo investigado desde los principios de los computadores compartidos. Sin embargo, es posible encontrar patrones comunes de uso que, por un u otro motivo, no son ideales para todos los casos:

### 1.3.1. Aplicaciones monolíticas con autorización ad hoc

En aplicaciones monolíticas hechas con frameworks MVC[2], es común que los controladores incluyan código que se encargue de la autorización, mezclado con el código relevante para la vista. Esto lleva consigo dos dificultades:

1. Mala separación de intereses, lo cual puede llevar a crear repetición, dificultando la mantenibilidad y la reutilización de componentes: Si se decide crear una nueva vista con información similar, es necesario repetir el código que se encarga de la autorización.
2. La autorización se suele implementar "desde cero" por personas que no son expertas en el tema, lo cual puede causar problemas de seguridad. Por ejemplo, un error común es el siguiente: Al intentar acceder a un recurso al que uno no tiene acceso, un servidor mal configurado podría responder un error HTTP 403 en lugar de 404, informando a usuarios no autorizados de que el recurso existe. Una base de datos puede no incluir el recurso en su respuesta, haciendo que la existencia o inexistencia de este sean indistinguibles.

[2]La arquitectura MVC, o Model View Controller, es una arquitecturas para desarrollar interfaces gráficas, en particular aplicaciones Web. Aunque existen variaciones, es la base de la gran mayoría de las arquitecturas en uso en aplicaciones monolíticas desarrolladas hoy en día.

Como solución a estos problemas, han surgido servicios de autorización, que se programan independientemente de las aplicaciones. Los servicios existentes se diferencian principalmente en los lenguajes que usan para definir reglas y en las arquitecturas con las que están desplegados en producción.

### 1.3.2. Lenguajes

Existen distintos lenguajes de dominio específico orientados a la autorización, que pueden servir para definir políticas de seguridad o relaciones entre entidades de la aplicación, dependiendo de cómo están destinados a usarse. A continuación se presentan dos lenguajes, uno que usa reglas para definir políticas de seguridad, y otro que lo hace creando listas de relaciones entre entidades.

#### Polar

Polar [12] es un lenguaje de autorización diseñado por Oso Security, Inc. Está orientado a escribir reglas para relacionar actores (como usuarios), permisos (como "puede leer") y recursos (objetos de la aplicación, como un documento). Las reglas se definen dependiendo de las características que tengan los recursos que se relacionan. Para definir estas reglas, puede usar funciones del lenguaje en que esté programado el *back-end*, como Python o Ruby.

El lenguaje viene con funcionalidad incluida para facilitar patrones comunes de uso, por ejemplo:

- Roles por recurso: Los roles son entes que relacionan actores con recursos (eg, administrador de una carpeta), que facilitan la asociación con permisos, es decir, la creación de reglas, como que todo administrador tiene permiso para eliminar objetos de la carpeta que administra. Además, permiten el desarrollo de funcionalidad adicional, como la que se ve en el siguiente punto.
- Herencia simple de roles: Es posible definir que al tener un rol automáticamente se obtienen los permisos de otro, lo cual facilita la creación de jerarquías de autorización. Es decir, es fácil definir que un "editor" de un documento debería poder hacer todo lo que un "lector" puede. Adicionalmente, los roles se pueden heredar de otros recursos: Se puede definir que un usuario con ciertos permisos para una carpeta tenga los mismos permisos para los archivos dentro de ella, o sus subcarpetas.
- Pattern Matching: Al definir reglas, se puede usar *pattern matching*, de modo que los grupos para los que las reglas aplican pueden ser conjuntos generales o específicos, o incluso un solo usuario.

#### Zanzibar

Zanzibar [20] es el sistema de autorización que usa Google en sus múltiples servicios. Su diseño ha influido en la industria, de modo que ahora AirBnB usa un sistema similar[3], y existen alternativas Open-Source como SpiceDB. [5]

El lenguaje de autorización de Zanzibar es menos flexible que Polar. Mientras que Polar ofrece al programador distintas maneras de definir las decisiones de autorización, Zanzibar solo permite el uso de un tipo de decisión, obligando al programador a adaptar su aplicación a este paradigma. El cómo lo hace se describe a continuación.

El uso de roles por recurso, opcional en Polar, es obligatorio en Zanzibar. Lo único que se puede definir en el archivo de configuración es la herencia de roles, tal como está descrita en el segundo punto de Polar.

A cambio de esta inflexibilidad, Zanzibar ofrece una base de datos adicional, optimizada para las operaciones de conjuntos y los recorridos en grafos, que son operaciones que debe hacer frecuentemente para las consultas de autorización. Esta base de datos que trae Zanzibar es distinta de la que usa la aplicación para su funcionamiento normal. El objetivo es que en ella se guarden los datos relevantes a las preguntas de autorización, siguiendo una estructura de tuplas definidas por Zanzibar.

El lenguaje no permite escribir reglas para cada aplicación, sino que se usa siempre la misma regla, que es muy general: una autorización tiene éxito si existe un camino desde el usuario hasta el recurso, a través del grafo definido por las tuplas en la base de datos asociada a la aplicación.

Este diseño facilita el control para muchos casos de uso, todos aquellos en los que la configuración de la autorización esté completamente separada del resto de la funcionalidad de la aplicación. Por ejemplo, en YouTube se pueden elegir las personas que pueden ver un video o hacerlo público, ambas opciones son fácilmente representables con tuplas de Zanzibar, y son independientes del resto de la funcionalidad importante de la aplicación. Por otro lado, en una aplicación de chat, los mensajes están relacionados con sus remitentes y destinatarios, y agregar esta información a la base de datos de Zanzibar probablemente duplique información que ya se puede encontrar en la de la aplicación.

### 1.3.3. Despliegues y Uso

#### Aplicaciones con un Lenguaje de Autorización Embebido

Una opción es la de usar un lenguaje de dominio específico de autorización para definir las reglas, y luego usarlo desde una aplicación creada en un lenguaje de propósito general (como Python), embebiendo el lenguaje de autorización en el de propósito general a través de una librería. El lenguaje Polar está diseñado para usarse de esta manera, ofrece *bindings*

[3]Sistema de autorización de AirBnB: `https://medium.com/airbnb-engineering/himeji-a-scalable-centralized-system-for-authorization-at-airbnb-341664924574`

para embeberlo en NodeJS, Python, Go, etc. Luego, cada vez que un usuario intenta hacer una acción, el código del *back-end* es responsable de llamar a la API del lenguaje embebido para determinar si este usuario está autorizado.

Este enfoque permite que, si una aplicación inicialmente monolítica migra a una arquitectura basada en microservicios programados en distintos lenguajes, las reglas de autorización sean hasta cierto punto reutilizables en los nuevos lenguajes. El grado en que son reutilizables las definiciones de las reglas depende del lenguaje y de las reglas en cuestión, ya que Polar permite usar funcionalidad del lenguaje anfitrión (como Python), lo cual limita su portabilidad al usarse desde otros lenguajes. Aunque no lo permitiera, la superficie de contacto entre el lenguaje anfitrión y el de autorización es considerable, ya que debe realizar las tareas de asociar los objetos de ambos lenguajes y verificar los permisos cuando sea necesario.

### Servicios Externos

En el caso de aplicaciones con escalas masivas, como Google o AirBnB, resulta conveniente externalizar la autorización a un servicio adicional. En el caso de Google, este servicio es Zanzibar, que mantiene y procesa listas de control de acceso (ACLs) de forma distribuida. Entonces, cada aplicación de Google debe hacer llamadas a los servidores de Zanzibar para que éste se encargue de la autorización, además de hacer llamadas a su base de datos (cualquier otra) como haría una aplicación común y corriente.

Un riesgo con esta estrategia es que la disponibilidad del servicio es crítica, i.e. es un punto único de fallo. Si el servicio de autorización es lento, todas las solicitudes a las aplicaciones serán lentas. Y si el servicio enfrenta una caída, las aplicaciones no pueden funcionar. Por este motivo, las aplicaciones que usan estas opciones suelen ser de empresas grandes. Google puede mantener un servicio de autorización disponible el 99.999 % del tiempo, y darle cientos de servidores para replicarlo alrededor del mundo, pero la gran mayoría de las empresas no tienen los recursos para hacerlo.

Esta forma de ser desplegado explica parte de las decisiones de diseño del lenguaje Zanzibar, en particular el hecho de que contenga una base de datos propia en lugar de integrarse con la de la aplicación. La integración significaría un mayor número de interacciones entre servidores, aumentando la latencia de cada solicitud.

### Bases de Datos

La mayoría de los motores de bases de datos tienen algún tipo de autorización incluida [2, 1]. Permiten crear usuarios, y asignarles lectura o edición en tablas, columnas, otros usuarios, etc. En algunos motores esta funcionalidad es limitada, pero hay otros que ofrecen más posibilidades.

Desde 2016 ([10]) en PostgreSQL se permite escribir reglas que definan el acceso y permisos en ciertos recursos, englobadas en un concepto llamado Row Level Security (RLS) [9, 8]. Con reglas RLS, se puede permitir a ciertos usuarios ver o modificar algunas filas en una tabla, pero otras no. Por ejemplo, en una aplicación de chat, se puede escribir una regla que permita

que los usuarios solo lean los mensajes que estén dirigidos a ellos.

Las reglas RLS se deben definir directamente en SQL. Este sistema se usa en Supabase [14], un "Backend-as-a-Service" (BaaS), servicio que permite definir el esquema de una base de datos y luego acceder a ella a través de una API Web, obviando la necesidad de implementar un *back-end* tradicional. Además de este BaaS, que integra varios servicios adicionales, las reglas RLS también se deben usar en alternativas autónomas como PostgREST y PostGraphile [19].

Sin embargo, se identifican tres clases de problemas con esta alternativa:

1. Las formas de definir permisos que son el estado del arte incluyen funcionalidad que no está disponible en SQL, como los roles por recursos y la herencia de estos roles, ya sea directamente o a través de otros recursos.
2. Las reglas se ejecutan automáticamente en SQL en vez de ser llamadas por un lenguaje de propósito general, lo que limita la flexibilidad alcanzable en la definición de permisos nuevos.
3. Usar SQL directamente no provee funcionalidad adicional relacionada a la autorización. Por ejemplo, no permite hacer fácilmente autorización en el *front-end*, lo que, aunque no provee seguridad, mejora la experiencia de usuario.

El propósito de este trabajo es eliminar los problemas del tipo 1, aumentando la ergonomía del uso de autorización en Postgres y acercándola a los estándares del estado del arte. Los problemas del tipo 2 son inherentes a la forma de funcionar de la autorización en bases de datos, por lo que solucionarlos está fuera del alcance de este trabajo. Por último, los de tipo 3 sí son solucionables, pero se descarta su inclusión en el trabajo de memoria ya que no son esenciales.

A cambio, el uso de autorización directa en el motor de bases de datos significa que las reglas se ejecutan automáticamente en cada consulta, lo cual tiene varias ventajas en comparación con las alternativas mencionadas anteriormente:

1. Si la aplicación tiene un *back-end*, este deja de ser una parte en la que se debe confiar, ya que los accesos a información del mismo *back-end* están regulados por las reglas de autorización. Así, se eliminan muchas clases de fuentes de errores posibles, como inyección SQL.
2. Cada conexión a la base de datos puede ver información distinta. Un caso de uso trivial de esto es que es posible simular funcionamiento *single-tenant* en despliegues *multi-tenant*[4].

[4]Por razones de seguridad, algunas empresas solicitan que las aplicaciones web de las que son clientes estén desplegadas en servidores independientes, distintos de los servidores que usa la misma aplicación para otros clientes. Este concepto se conoce como *single-tenancy,* en contraposición con la *multi-tenancy* que es cuando se usa la misma infraestructura para ambos clientes. La base de datos, separando la información de ambos clientes, puede hacer que la *multi-tenancy* sea indistinguible de la *single-tenancy,* ignorando diferencias de rendimiento.

3. No hay superficie de contacto entre el lenguaje de autorización y el lenguaje en que se implementa el *back-end*, en caso de que este último exista. Migrar partes de la aplicación de un lenguaje a otro se puede hacer sin tener que portar partes del sistema de autorización.

Dados estos beneficios, es deseable que exista un sistema de autorización diseñado para funcionar en la base de datos, pero que no tenga las desventajas que involucran las reglas en SQL. En el siguiente capítulo se detalla esta propuesta.

# Capítulo 2

# Visión General del Trabajo

Resumiendo lo que se ha dicho, el problema de la autorización tiene muchas soluciones para distintos casos de uso. Para PostgreSQL, se determina quién puede acceder a cada entidad[1] usando `POLICY`s, descripciones escritas en SQL. Es una forma cómoda de definir reglas, que simplifica el despliegue de la aplicación, pero no contiene la funcionalidad de los lenguajes del estado del arte de la aplicación. Entonces, Pilpilang busca ser un lenguaje más moderno para definir reglas, con expresividad y características similares a las de lenguajes del estado del arte, pero tal que las aplique usando el mecanismo existente de Postgres.

Pilpilang estará entonces diseñado para que sus reglas se puedan *convertir* en descripciones escritas en SQL[2], ya que estas últimas se pueden introducir a la base de datos. Cada permiso tendrá como resultado un valor Booleano, y determinará si un usuario dado puede ejecutar alguna de las 4 operaciones principales de SQL[3] en un recurso.

Se presenta a continuación una introducción a las funcionalidades de Pilpilang. Un "programa" Pilpilang consiste en recursos, actores, permisos y funciones que los relacionan. En este sentido, su estructura es similar a la que se suele encontrar en lenguajes de autorización de reglas como Polar.

## 2.1. Recursos

Un recurso es un ente sensible de la aplicación, al que se quiere agregar autorización. El caso más simple sería una fila de una tabla de la base de datos, que solo algunos usuarios deberían poder ver.

Los recursos se declaran especificando la tabla a la que hacen referencia, su llave primaria y las columnas relevantes para definir reglas de seguridad:

[1] En rigor, a cada fila de la base de datos.

[2] Esto significa que la implementación de Pilpilang es un *transpilador*, ya que "compila" de un lenguaje a otro lenguaje (SQL).

[3] INSERT, SELECT, UPDATE, DELETE

```
resource Document {
    table "documents"
    key ["id"]
    columns [
        title: String,
        contents: String,
    ]
}
```

Al leer esto, Pilpilang sabe que cada `Document` tiene un campo "title" y un campo "contents". Si estos campos se usan posteriormente para definir reglas, Pilpilang usará esta información para chequear que las columnas son usadas correctamente.

## 2.2. Actores

Los actores son una pequeña generalización de los usuarios. Son un ente que es o no autorizado a acceder a un recurso. Se configura de un modo similar a un recurso, asociándolo a una tabla en la base de datos y agregando configuraciones adicionales en su bloque:

```
actor User {
    table "users"
    key ["user_pk"]
    columns [
        age: Int,
        username: String
    ]
}
```

Se permite tener más de un tipo de actor ya que una aplicación podría tener distintas reglas de acceso para estos tipos, por ejemplo separando usuarios normales, auditores, inspectores y administradores.

## 2.3. Permisos

Los permisos son lo más importante en un programa Pilpilang. Se definen en base a predicados (funciones que se evalúan a Verdadero o Falso) cuyos argumentos son un Recurso y un Actor. Estos determinan capacidades que pueden o no tener los actores, y derivan de las operaciones que se pueden hacer en SQL. Actualmente, los permisos que hay son `can_select`, `can_update`, `can_insert`, `can_delete` y `can_anything`.

Una vez que se define un permiso, es posible traducirlo a SQL. Por ejemplo, el siguiente permiso permite ver videos con edad restringida:

```
can_select(u: User, v: Video)
    if v.age_restricted = false || u.age > 18
```

se traduce a[4]

```
CREATE POLICY policy1 ON videos
FOR SELECT
WITH CHECK (current_user IN
(SELECT actorNick.id FROM users actorNick
WHERE videos.age_restricted = False
OR actorNick.age > 18));
```

No es difícil notar que la traducción es larga y compleja comparada con el permiso de Pilpilang, mostrando el poder de usar un DSL de autorización en vez de usar extensiones de SQL. Sin embargo, hay formas más cortas de escribir un permiso equivalente en SQL. Si se busca una comparación más justa, donde ambas versiones estén escritas por humanos, ir al capítulo 4.

## 2.4. Funciones

Las funciones son una abstracción que generaliza a los permisos, y son convenientes para definirlos de una manera más modular. Mientras que los permisos disponibles están limitados a `can_select`, `can_update` y similares, con un recurso y un actor como argumentos, una función puede tener cualquier nombre y cualquier número de argumentos. Luego, se pueden usar en los predicados de los permisos, o de otras funciones.

Ejemplo:

```
can_update(u: User, doc: Document)[act_doc: User_Doc]
    if act_doc.collaborator = u && act_doc.doc = doc
        && is_owner(act_rec)

is_owner(act_doc: User_Doc) if act_doc.relationship_type = "owner"
```

El ejemplo muestra varias funcionalidades de Pilpilang. La primera es que las funciones se pueden usar desde otras definiciones, ya que la definición de `can_update` usa `is_owner`. También muestra que existe la igualdad entre dos `User`s, como se hace en `act_doc.collaborator = u`, la cual se hará comparando las llaves en SQL. Otros tipos que se pueden comparar son recursos mediante sus llaves, *strings*, números o *booleans*.

[4]El uso de la variable current_user es una simplificación que se hace por legibilidad. En la mayoría de los casos no es lo que se desea, la alternativa correcta depende de la configuración del servidor, como por ejemplo `auth.user()` para Supabase, o una compleja consulta JSONB en PostgREST. En todo caso esto es un detalle ortogonal al diseño de Pilpilang.

El permiso `can_update` del ejemplo anterior tiene otra cosa que puede ser llamativa: un argumento implícito. El valor `act_rec` se refiere a una tabla que combina entidades `User` y `Document`, como suele ser necesario para representar relaciones *many-to-many* en bases de datos relacionales. El predicado de `can_update` será exitoso si *existe* un `act_rec` que cumpla con el predicado. Para que esto funcione, la declaración de `User_Doc` sería similar a:

```
resource User_Doc {
    table "users_documents"
    columns [
        # collaborator es una propiedad de User_doc cuyo tipo
        # es User, y en la base de datos la columna user_id
        # es la llave foránea apuntando al User correspondiente.
        collaborator: User (user_id),

        # análogo a lo anterior
        doc: Document (document_id),

        relationship_type: String,
    ]
    # la llave primaria es compuesta
    keys ["user_id", "document_id"]
}
```

Lo que hace el ejemplo es, entonces, buscar una fila que relacione usuarios con documentos, que cumpla con que la relación esté marcada como "owner".

## 2.5. Comentarios Adicionales

El ejemplo anterior no es la forma más natural de manipular relaciones *many-to-many* entre actores y recursos en Pilpilang, se hizo de ese modo solo para demostrar el uso de un argumento implícito. Una forma más eficiente de obtener el mismo resultado sería definir `User_Doc` como un `actor` y usarlo como argumento explícito en `can_update`.

En el contexto de Pilpilang, se suele usar la palabra "reglas" para referirse a permisos y funciones conjuntamente.

Eso finaliza la visión general, el capítulo a continuación es una formalización más rigurosa.

# Capítulo 3

# Solución

En este capítulo, se detalla más profundamente el diseño y funcionamiento de Pilpilang, formalizando en mayor medida los resultados que genera. Si busca un tutorial simple, se sugiere consultar el capítulo anterior.

La implementación del lenguaje se realizó en Haskell, ya que es un buen lenguaje para escribir transpiladores, como han demostrado los proyectos Pandoc y Elm, que también son transpiladores escritos en Haskell. El código del proyecto se puede encontrar en el repositorio `https://github.com/vichoreyes/pilpilang`

La estructura de la implementación se subdivide en tres partes. En primer lugar está la sintaxis, que describe en gramática BNF cómo se escriben los programas Pilpilang. En segundo lugar se presenta el sistema de tipado, ya que el lenguaje usa tipos estáticos en el momento de compilación, para validar las reglas y evitar emitir SQL incorrecto. Por último, está la descripción y funcionamiento de las conversiones finales a SQL, un tema complejo que se estructura en varias partes.

## 3.1. Sintaxis

En esta sección se muestra la sintaxis de Pilpilang usando notación de gramáticas BNF [5]. Aunque hay formas de convertir estas gramáticas en ejecutables, como Happy[6], la implementación fue realizada con *parser combinators* [17] debido a que se integran más nativamente con el resto del compilador. Son una manera de escribir *parsers* que ha demostrado sus capacidades, siendo usada por proyectos como Idris y Dhall [4, 7]. En la implementación, el módulo encargado de la sintaxis se encuentra en `src/Syntax.hs`, y la librería específica que se usa es `megaparsec` [16].

Las componentes principales de un archivo Pilpilang son las reglas, pudiendo ellas ser permisos o funciones. Estas reglas son $a$ en la gramática siguiente:

| | | | |
|---|---|---|---|
| $x$ | $\in$ | Variables | |
| $c$ | $\in$ | Expresiones Literales | |
| $t$ | $\in$ | Tipos | Int, String, tipos definidos por el usuario... |
| $f$ | $\in$ | Identificadores | Nombres de funciones |
| $v$ | $::=$ | $x \mid c \mid v.x$ | Valores: Variables, constantes o accesos |
| $a$ | $::=$ | $h$ if $p$ | Asociaciones: Cabecera y Predicado |
| $\pi$ | $::=$ | can_select \| can_update | Permisos otorgables |
| | $\mid$ | can_insert \| can_delete \| can_anything | |
| $p$ | $::=$ | | Un predicado puede ser: |
| | | $p \wedge p$ | AND de predicados |
| | $\mid$ | $p \vee p$ | OR de predicados |
| | $\mid$ | $v = v$ | Comparación de valores |
| | $\mid$ | $v < v$ | |
| | $\mid$ | $v > v$ | |
| | $\mid$ | $f(\overline{v_i})$ | Llamada a una función |
| $h$ | $::=$ | | Una cabecera puede ser: |
| | | $\pi(x_1 : t_1, x_2 : t_2)[\overline{x_j : t_j}]$ | de un permiso |
| | $\mid$ | $f(\overline{x_i : t_i})[\overline{x_j : t_j}]$ | de una función |
| | $\mid$ | $\pi(x_1 : t_1, x_2 : t_2)$ | Los anteriores sin argumentos implícitos: |
| | $\mid$ | $f(\overline{x_i : t_i})$ | ie, azúcar sintáctica para que estén vacíos |

Adicionalmente, existe una sintaxis para definir entidades, siendo éstas actores o recursos, que luego se pueden usar como nuevos tipos en la definición de permisos y funciones. Esto se logra con la gramática:

$$\begin{array}{rcl} \varepsilon & ::= & (\text{actor} \mid \text{resource})\ t\ \{\overline{k : k'}\} \\ k & ::= & \text{table} \mid \text{columns} \mid \text{key} \end{array}$$

Los contenidos de $k'$ dependen de los de $k$

| $k$ | $k'$ | Explicación |
|---|---|---|
| table | $q$ | La tabla en base de datos que contiene las entidades $t$. |
| columns | $[\overline{x_i : t'_i}]$ | Las columnas de la tabla $t'$ que se usen en las reglas, y sus tipos. |
| keys | $[\overline{pk_i}]$ | La o las columnas de la tabla que, conjuntamente, sean llave primaria. |

Nótese que en "columns", los tipos no son $t$ sino $t'$, una sutil diferencia: $t'$ requiere que al escribir un tipo no primitivo, se agregue su llave o llaves foráneas entre paréntesis:

$$t' ::= \mathit{Int} \mid \mathit{String} \mid \mathit{Boolean} \mid t(\overline{fk_i})$$

Un ejemplo de $\varepsilon$ podría ser la siguiente definición de recurso:

```
resource Sala {
    # En la base de datos, existe la tabla "salas"
    table "salas"

    # Los num_sala se pueden repetir en distintos edificios,
    # así que el edificio es parte de la llave primaria
    key ["num_sala", "id_edificio"]

    columns [
        sillas: Int,
        ubicacion: String,
        propietario: Persona (id_propietario),
    ]
}
```

## 3.2. Sistema de tipado

El lenguaje Pilpilang usa un sistema de tipado estático para asegurarse de que los programas estén correctamente formados. De este modo, se asegura de que no se esté generando código SQL basura. Este sistema está estructurado conceptualmente con cuatro fases distintas, aunque en la práctica se pueden fusionar en solo dos.[1]. La implementación, que se encuentra en `src/Types.hs` en el repositorio, sí fusiona las cuatro fases en dos. En esta sección se explicarán reglas de tipado. Cuando exista una función en la implementación análoga a una regla, se mencionará su nombre.

### 3.2.1. Tipos

Primero se recolecta la información de las entidades: Los actores se guardan en el ambiente $\Delta$ y los recursos en el ambiente $\Sigma$. Ya que se usan con frecuencia juntos, se define $\Omega = \Delta \cup \Sigma$. La información que se guarda para cada entidad consiste en:

- El nombre de la entidad
- Los nombres de sus columnas
- La tabla a la que hacen referencia en la base de datos
- La(s) llave(s) primaria(s)

Revisar la validez de los tipos de las columnas no se hace todavía, se revisará en la tercera fase.

[1] Un compilador de una sola fase crea limitaciones como la necesidad de usar *forward declarations*: `https://en.wikipedia.org/wiki/Multi-pass_compiler`

### 3.2.2. Cabeceras

Luego se revisan las cabeceras de las asociaciones, partiendo por los permisos. Para ellos, se sigue la siguiente regla de inferencia, que indica que el primer argumento debe ser un actor, el segundo un recurso, y los argumentos implícitos pueden ser cualquiera de esas dos opciones:

$$\frac{t_1 \in \Delta \qquad t_2 \in \Sigma \qquad t_j \in \Delta \cup \Sigma}{\Delta, \Sigma \vdash \pi(x_1 : t_1, x_2 : t_2)[\overline{x_j : t_j}]}$$

Las funciones son similares a los permisos. Una diferencia es que cualquier tipo de argumento explícito está permitido, y puede haber tantos como se quiera. Sin embargo, los argumentos implícitos no pueden ser de tipos primitivos, ya que serán usados como cuantificadores universales. Además, al revisar los tipos de las cabeceras de funciones, se crea el mapa $\Lambda(\cdot)$, que asocia los nombres de las funciones con los tipos de sus argumentos explícitos, para uso futuro:

$$\frac{t_i \in \Delta \cup \Sigma \cup [\text{Int, String, Bool}] \qquad t_j \in \Delta \cup \Sigma}{\Delta, \Sigma \vdash \ f(\overline{x_i : t_i})[\overline{x_j : t_j}] \ \dashv \Lambda(f) := \overline{t_i}}$$

En la implementación, la función que realiza el fuerte de estas tareas es `mkLocalVars`, aunque también son relevantes `mkKey` y `mkHeader`.

### 3.2.3. Columnas

Esta fase simplemente se asegura de que existan los tipos de cada columna de cada entidad. Es decir, en una definición así:

```
resource Event {
    table "events"
    keys ["event_id"]
    columns [
        creator: User(creator_id),
        description: String,
        ...
    ]
}
```

esta fase se encarga de revisar que `User` y `String` existan, y que la cantidad de llaves foráneas en `creator` sea la misma que la cantidad de llaves primarias en la definición de `User`.

Formalmente, esto significa que una declaración de actor o recurso es válida si todos los tipos de sus columnas son válidos:

$$\frac{\Delta, \Sigma \vdash_{t'} \ t'_i}{\Delta, \Sigma \vdash \ (\text{actor}|\text{resource}) \ t \ \{columns : [\overline{x_i : t'_i}]\}}$$

Un tipo de columna es válido si es un tipo primitivo:

$$\frac{t \in \{\text{Int, String, Bool}\}}{\Delta, \Sigma \vdash_{t'} \ t}$$

O es válido si está definido en el programa y tiene la misma cantidad de llaves primarias que la cantidad de llaves foráneas referenciándolo:

$$\frac{t \in \Omega \wedge |\Omega(t).keys| = |fk_i|}{t(\overline{fk_i})}$$

Al terminar esta fase, $\Sigma(\cdot)$.columns y $\Delta(\cdot)$.columns son capaces de asociar los nombres de columnas con sus tipos respectivos.

En la implementación, las funciones de `Types.hs` que realizan las tareas de esta fase son `genEnt` y `genType`, revisando la entidad completa y los tipos de las columnas respectivamente.

### 3.2.4. Predicados

Una vez que se han realizado las fases anteriores, se tiene información sobre la aridad y tipos de cada función (guardadas en $\Lambda$, obtenidas en 3.2.2), y las columnas de cada tipo (guardados en $\Omega = \Delta \cup \Sigma$, obtenidas en 3.2.3). Con esto, se puede revisar los tipos de los predicados del programa:

$$\frac{\Omega, \Lambda, \Gamma \vdash_p p}{\Lambda \vdash f(\Gamma_1)[\Gamma_2] \ \textit{if} \ p}$$

Nótese que en la regla anterior, el ambiente $\Gamma = \Gamma_1 \cup \Gamma_2$ proviene de los argumentos de $f$.

$$\frac{\Omega, \Lambda, \Gamma \vdash_p p_1 \qquad \Omega, \Lambda, \Gamma \vdash_p p_2}{\Omega, \Lambda, \Gamma \vdash_p p_1 \wedge p_2}$$

$$\frac{\Omega, \Lambda, \Gamma \vdash_p p_1 \qquad \Omega, \Lambda, \Gamma \vdash_p p_2}{\Omega, \Lambda, \Gamma \vdash_p p_1 \vee p_2}$$

$$\frac{\Omega, \Gamma \vdash_v v_1 : t_1 \qquad \Omega, \Gamma \vdash_v v_2 : t_2 \qquad t_1 = t_2}{\Omega, \Lambda, \Gamma \vdash_p v_1 = v_2}$$

$$\frac{\Omega, \Gamma \vdash_v v_1 : \text{Int} \qquad \Omega, \Gamma \vdash_v v_2 : \text{Int}}{\Omega, \Lambda, \Gamma \vdash_p v_1 < v_2}$$

$$\frac{\Omega, \Gamma \vdash_v v_1 : \text{Int} \qquad \Omega, \Gamma \vdash_v v_2 : \text{Int}}{\Omega, \Lambda, \Gamma \vdash_p v_1 > v_2}$$

$$\frac{\forall i. \Omega, \Gamma \vdash_v v_i : t'_i \qquad \Lambda(f) = \overline{(t_i)} \qquad \forall i. t'_i = t_i}{\Omega, \Lambda, \Gamma \vdash_p f(\overline{v_i})}$$

A su vez, se infieren los tipos de los valores con las siguientes reglas:

$$\frac{c \text{ literal}}{\Omega, \Gamma \vdash_v c : \mathit{type}(c)}$$

$$\frac{\Gamma(x) \text{ está definido}}{\Omega, \Gamma \vdash_v x : \Gamma(x)}$$

Considerando que $\Omega = \Sigma \cup \Delta$, un mapa que asocia nombres de tipos con su información definida previamente,

$$\frac{\Omega, \Gamma \vdash_v v : t \qquad (a : t') \in \Omega(t).\text{columnas}}{\Omega, \Gamma \vdash_v v.a : t'}$$

Las reglas de checkeo de tipos que se vieron aquí, $\Omega, \Lambda, \Gamma \vdash_p$ y $\Omega, \Gamma \vdash_v$, se implementan mediante las funciones `cPredicate` y `cValue` respectivamente.

## 3.3. Conversiones

Por último, corresponde convertir las asociaciones a reglas en SQL. El proceso para realizarlo es complejo y se describe a continuación. Si se desea encontrar la implementación de este proceso, se puede encontrar en el repositorio del proyecto, en el archivo `src/Conversion.hs`. Sin embargo, la separación de tareas descrita en esta sección no es tan limpia en la implementación, por lo que puede ser difícil de entender, más que los módulos de sintaxis y tipado.

### 3.3.1. Motivación

El proceso de convertir asociaciones a SQL es la última parte, y el resultado obtenido depende de la asociación transformada de múltiples maneras. Por ejemplo, una función como la siguiente:

```
abc(a: T, b:T) if a.some_int = b.other_int
```

Se transformará simplemente en

```
SELECT a.pk, b.pk FROM T_table a, T_table b WHERE a.some_int = b.other_int
```

Pero si las columnas del ejemplo anterior fueran referencias a otras tablas, la consulta final tendría que involucrarlas también:

```
abc(a: T, b:T) if
    a.some_driver.miles_traveled
    = b.other_driver.miles_traveled
```

Se transforma en

```
SELECT a.pk, b.pk
FROM T_table a , T_table b, drivers adriver, drivers bdriver
WHERE adriver.miles_traveled = bdriver.miles_traveled
AND a.driver_key = adriver.pk
AND b.driver_key = bdriver.pk
```

Consulta que, como se puede observar, requiere visitar la tabla "drivers" además de las tablas dadas por los argumentos de la función. Además, para hacer los JOINs apropiados, requiere agregar condiciones adicionales al WHERE.

### 3.3.2. Diseño

La traducción de cada asociación tendrá como resultado cuatro estructuras:

1. La lista de tablas involucradas en la consulta (lo que va en el FROM), junto con sus apodos. Los apodos son importantes porque permiten distinguir entre distintas filas de la misma tabla.
2. La lista de columnas que se seleccionarán de las tablas mencionadas. Estas columnas siempre serán las llaves primarias de las tablas referenciadas por la cabecera de la función.
3. Las condiciones de igualdad de llaves foráneas que permiten hacer cruces con otras tablas, cuando el predicado contiene indirecciones.
4. La traducción directa del predicado de la asociación.

Estos elementos se combinan de una manera que depende de si lo que se está transformando es una función o permiso:

Si es una función, está siendo llamada por otro predicado, por lo que se debe crear una consulta que cumpla con la convención de llamadas de Pilpilang. La convención dicta que se genere una tabla con todas las llaves primarias de los argumentos, resultando entonces en algo así:

```
SELECT <llaves primarias, del punto 2 de la lista>
FROM <tablas con sus apodos, del punto 1>
WHERE <condiciones del punto 3>
AND <condición del punto 4>
```

Por otro lado, puede ser un permiso en lugar de una función. Los permisos deben traducirse en una POLICY de Row-Level-Security[8] de Postgres, la cual tendrá la siguiente forma:

```
CREATE POLICY <nombre aleatorio>
ON <tabla del recurso del permiso, del punto 2>
FOR <operación del permiso, eg, INSERT>
WITH CHECK (current_user IN (
SELECT <llave primaria del actor del permiso, del punto 2>
FROM <tablas con sus apodos, del punto 1>
WHERE <condiciones del punto 3>
AND <condición del punto 4>
));
```

Se describió el uso de los cuatro puntos de la lista para crear las consultas y policies postgres, pero no se ha descrito cómo se obtendrán esos puntos. A continuación se describirá de dónde se obtienen.

### Lista de columnas para seleccionar

La lista de columnas que se seleccionarán en la consulta final es lo más simple, solo depende de la cabecera de la función o permiso.

Cada argumento *explícito* de la función tendrá un tipo asociado, $t$. Y ya sabemos que $t$ es válido porque pasamos por la fase de revisar tipos, que representa una entidad definida previamente. Cada entidad que se define contiene su llave primaria, representada por una lista de columnas. Estas columnas son las que se seleccionarán, todas ellas concatenadas.

| Cabecera | Columnas |
|---|---|
| $f(x_1 : t_1, x_2 : t_2)[x_3 : t_3]$ | $[\Omega(t_1).keys,\ \Omega(t_2).keys]$ |

Donde, al igual que en la sección anterior, $\Omega = \Sigma \cup \Delta$ representa la información de cada tipo.

### Lista de tablas involucradas, junto con sus apodos

Tal como la lista de columnas, se parte analizando la cabecera de la función, agregando una tabla por cada argumento, *incluyendo los argumentos implícitos*:

| Cabecera | Tablas |
|---|---|
| $f(x_1 : t_1, x_2 : t_2)[x_3 : t_3]$ | $[\Omega(t_1).table,\ \Omega(t_2).table,\ \Omega(t_3).table]$ |

Sin embargo, tal como se describió en Motivación, un predicado con referencias a otras tablas necesita agregarlas a la consulta. Por este motivo, cuando se está recorriendo el predicado, se buscan los valores $v.c$ donde $\Omega, \Gamma \vdash_v v.c : t \ \wedge \ t \notin \{\text{Int, String, Bool}\}$, y por cada uno de ellos se agrega $t.table$ a la lista de tablas.

Los apodos se generan aleatoriamente, asegurándose de que no se repitan.

### Lista de condiciones de JOIN

En la subsección previa se describió que puede ser necesario agregar tablas adicionales a la consulta, cuando hay valores $v.c$ donde $\Omega, \Gamma \vdash_v v.c : t \ \wedge \ t \notin \{\text{Int, String, Bool}\}$. Cada vez que se agrega una de estas tablas, se debe agregar también una condición a la consulta, para cruzar las filas adecuadas. La condición será simplemente la igualdad de las llaves foráneas de una tabla con las llaves primarias de la otra.

A continuación se muestra un ejemplo de estas condiciones:

```
resource A {
    table: "tab_a"
    keys: [key1, key2]
    columns: [
        some_bool: Boolean
    ]
}

resource B {
    table: "tab_b"
    columns: [
        ref_a: A (a_fk1, a_fk2)
    ]
}

has_bool(b: B) if b.ref_a.some_bool
```

La función `has_bool` contiene `b.ref_a`, por lo que se deben cruzar esas dos tablas. La condición generada depende entonces de las llaves de estas tablas, quedando[2]:

[2]Por simplicidad no se usan apodos en este ejemplo, pero en el sistema real sí se usan.

```
(tab_b.a_fk1, tab_b.a_fk2) = (tab_a.key1, tab_a.key2)
```

Esa condición se agregará a la consulta final, para hacer el cruce apropiado.

### Traducción directa del predicado

Los predicados se convierten a SQL, usando las reglas definidas a continuación:

$$\frac{p_1 \to_p P_1 \qquad p_2 \to_p P_2}{p_1 \wedge p_2 \to_p P_1 \text{ AND } P_2}$$

$$\frac{p_1 \to_p P_1 \qquad p_2 \to_p P_2}{p_1 \vee p_2 \to_p P_1 \text{ OR } P_2}$$

$$\frac{v_i \to_v V_i \qquad f \to_f Q}{f(\overline{v_i}) \to_p (\overline{V_i}) \text{ IN}(Q)}$$

donde $\to_f$ representa la conversión completa de otra asociación, es decir, el proceso completo recursivo que se está describiendo en esta sección.

$$\frac{v_1 \to_v V_1 \qquad v_2 \to_v V_2}{v_1 = v_2 \to_p V_1 = V_2}$$

Habiendo descrito las conversiones de los distintos tipos de predicados, se describen a continuación las conversiones de los valores ($\to_v$).

Los literales se representan igual en SQL.

$$\frac{}{c \to_v c}$$

Los accesos a campos primitivos de entidades se dejan como accesos a columnas de sus respectivas tablas. ($\to_t$ convierte un valor en su tabla)

$$\frac{\Omega, \Gamma \vdash_v v.c : t \qquad t \in \{\text{Int, String, Bool}\} \qquad v \to_t V}{v.c \to_v V.c}$$

El resto de los valores no son primitivos, por lo que para hacer comparaciones será necesario usar sus llaves primarias. Por este motivo se obtiene la representación y se agregan las llaves primarias.

$$\frac{\Omega, \Gamma \vdash_v v : t \qquad t \notin \{\text{Int, String, Bool}\} \qquad v \to_t V}{v \to_v V.pk}$$

Las variables se buscan en el ambiente apodos-tablas(), cuya creación se detalló hace dos subsecciones.

$$\frac{\text{apodos-tablas}(x) = X}{x \to_t X}$$

Los accesos a campos no primitivos de entidades, es decir, el acceso a otras entidades, involucran un cruce con otra tabla. En primer lugar, se busca si el valor en cuestión se encuentra en apodos-tablas($\cdot$). En caso contrario, se crean la tabla, apodo y condición nuevas, como se describió anteriormente, insertando el apodo nuevo en apodos-tablas($\cdot$). Luego se puede usar este apodo para la conversión del valor.

$$\frac{\Omega, \Gamma \vdash_v v.c : t' \qquad t' \notin \{\text{Int, String, Bool}\} \qquad \text{apodos-tablas}(v.c) = V'}{v.c \to_t V'}$$

Así, se han obtenido los cuatro puntos de la lista que se encuentra en 3.3.2. Como se menciona a continuación, con estos cuatro puntos se pueden generar las instrucciones SQL completas, con lo cual queda descrita la conversión de Pilpilang a SQL.

### 3.3.3. Ejemplo

Se tiene la base de datos representada por el diagrama 3.1, que describe las relaciones entre las tablas `users`, `messages` y `chats`. Para encargarse de la autorización, se usará el siguiente programa Pilpilang:

```
actor User {
    table "users"
    keys ["u_id"]
    columns [
        alcohol_ppm: Int,
    ]
}

resource Message {
    table "messages"
    keys ["m_id"]
    columns [
        chat: Chat (chat_id),
        contents: String,
    ]
}

resource Chat {
    table "chats"
    keys ["chat_id"]
    columns [
```

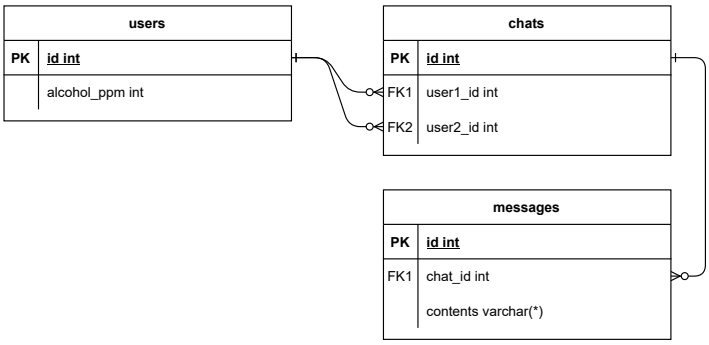


Figura 3.1: Diagrama de la base de datos del ejemplo

```
        user1: User (user1_id),
        user2: User (user2_id),
    ]
}

can_insert(u:User, m:Message) if
    is_not_drunk(u)
    && chat_contains(m.chat, u)

is_not_drunk(u: User) if u.alcohol_ppm < 5

chat_contains(c: Chat, u: User) if u = c.user1 || u = c.user2
```

El ejemplo se tratará de la traducción de `chat_contains`. El resultado es una consulta que obtiene todas las filas que cumplen la condición.

Ya que la cabecera tiene los argumentos `(c: Chat, u: User)`, esas son las variables del estado inicial, con tablas y apodos. También serán las llaves que se seleccionarán finalmente.

| apodos-tablas(·) | Tablas |
|---|---|
| c → c1 | chats c1 |
| u → u1 | users u1 |

Luego se debe recorrer el predicado, que es un OR de dos igualdades. La primera igualdad es `u = c.user1`. La variable `u` se reemplaza por su apodo (`u1`) junto con su llave primaria, quedando `u1.u_id`. El valor `c.user1` es de tipo User, y no se encuentra en la tabla de apodos actual, por lo que se debe crear un nuevo apodo y con él una nueva condición de JOIN. Se expande entonces el estado, quedando:

| apodos-tablas(·) | Tablas | Condiciones |
|---|---|---|
| c → c1 | chats c1 | |
| u → u1 | users u1 | |
| c.user1 → u2 | users u2 | `c1.user1_id = u2.u_id` |

La segunda parte del OR es análoga, por lo que quedará finalmente

| apodos-tablas(·) | Tablas | Condiciones |
|---|---|---|
| c → c1 | chats c1 | |
| u → u1 | users u1 | |
| c.user1 → u2 | users u2 | `c1.user1_id = u2.u_id` |
| c.user2 → u3 | users u3 | `c1.user2_id = u3.u_id` |

La traducción directa del predicado `u = c.user1 || u = c.user2` a SQL, siguiendo las reglas descritas anteriormente, será `u1.u_id = u2.u_id OR u1.u_id = u3.u_id`. Juntando esta conversión con las condiciones, tablas, apodos y columnas de selección, el resultado final será:

```
SELECT c1.chat_id, u1.u_id
FROM chats c1, users u1, users u2, users u3
WHERE
    u1.u_id = u2.u_id OR u1.u_id = u3.u_id
AND
    c1.user1_id = u2.u_id
AND
    c1.user2_id = u3.u_id
```

Esta consulta es una conversión adecuada para la regla, al ejecutarla obtiene todas las entidades que cumplen con el predicado. De este modo, resulta útil para insertarla en otras reglas o permisos.

# Capítulo 4

# Evaluación

Para evaluar la aplicabilidad y expresividad de Pilpilang, se observarán 3 ejemplos existentes de reglas de aplicaciones en SQL, averiguando si Pilpilang tiene la capacidad de representarlos en su estado actual. Los ejemplos de autorización son cortesía de Supabase [14], y se pueden encontrar en `https://github.com/supabase/supabase/tree/master/examples`.

Los ejemplos son:

1. Una aplicación de lista de tareas pendientes simple, en la que cada usuario tiene su propia lista y puede manipularla, sin que hayan formas de compartir listas o ítems.
2. La gestión de perfiles de usuarios en una aplicación de tipo red social, con nombres completos y nicknames, direcciones de sitios web y fotos de perfil.
3. Un clon simplificado de la aplicación de chat Slack, para la cual cada usuario tiene roles dados, puede enviar mensajes a través de canales, etc.

La complejidad de las reglas del tercer ejemplo es significativamente mayor que la del segundo, que a su vez es más complejo que el primero, por lo que este análisis permite identificar problemas con la aplicabilidad de Pilpilang a medida que la complejidad aumenta.

Las reglas de autorización de los ejemplos en SQL están mezcladas con algunas operaciones que no son específicamente de autorización, por lo que se omiten. Los ejemplos completos se pueden ver en el Anexo.

## 4.1. Lista de tareas pendientes

Este ejemplo se trata de una aplicación web multi usuario, en la que cada usuario tiene asociada una lista de tareas pendientes. Las tareas tienen asociado un texto que las describe, una referencia al usuario que las creó y un valor *booleano* que dice si se han completado o no. Cada usuario, y solo ese usuario, puede crear, ver, modificar y eliminar sus tareas. En la base de datos, estas tareas están en la tabla `todos`.

En SQL, estas reglas se pueden expresar de la siguiente manera:

```
alter table todos enable row level security;

create policy "Individuals can create todos." on todos for
    insert with check (auth.uid() = user_id);

create policy "Individuals can view their own todos. " on todos for
    select using (auth.uid() = user_id);

create policy "Individuals can update their own todos." on todos for
    update using (auth.uid() = user_id);

create policy "Individuals can delete their own todos." on todos for
    delete using (auth.uid() = user_id);
```

### 4.1.1. Equivalente Pilpilang

En Pilpilang, se expresarían de la siguiente manera:

```
actor User {
    table "auth.users"
    keys ["id"]
}

resource Task {
    table "todos"
    columns [
        user: User(user_id)
    ]
}

can_insert(u: User, t: Task) if u = t.user
can_select(u: User, t: Task) if u = t.user
can_update(u: User, t: Task) if u = t.user
can_delete(u: User, t: Task) if u = t.user
```

Para este caso de uso, sí se observa que el lenguaje es capaz de aplicarse al problema y solucionarlo.

### 4.1.2. Otros lenguajes de autorización

En la sección 1.3, se describieron los sistemas de autorización Polar y Zanzibar. Las reglas que se presentan a continuación no son equivalentes a las de Postgres y Pilpilang, ya

que tienen APIs de uso distintas, pero la funcionalidad prevista es la misma.

### Lista de tareas pendientes en Polar

En Polar, las reglas se expresan:

```
actor User {
}

resource Task {
    relations = {
        user: User
    };
}

allow(u: User, "create", t: Task) if u = t.user
allow(u: User, "read", t: Task) if u = t.user
allow(u: User, "update", t: Task) if u = t.user
allow(u: User, "delete", t: Task) if u = t.user

# o alternativamente, si son todas las acciones posibles:
# allow(u: User, _action, t: Task) if u = t.user
```

Y luego, en el *back-end* se deben hacer cumplir estas reglas manualmente:

```
def change_task(user, task, new_contents):
    if polar.authorize(user, "update", task):
        # autorizado exitosamente
        task.change(new_contents)
    else:
        raise AuthorizationError
```

Se puede ver que la solución en Polar es muy similar a la de Pilpilang, con la diferencia de que Pilpilang se integra con la base de datos, mientras que Polar se debe integrar con el *back-end* manualmente.

### Lista de tareas pendientes en Zanzibar

En Zanzibar el lenguaje es menos relevante, ya que las reglas que se siguen son siempre muy similares: encontrar un camino desde el actor al recurso a través del grafo de autorización de la aplicación. Lo relevante es agregar cada tarea a la base de datos de Zanzibar cuando se crea, y aprovechar eso para autorizar:

```
def create_task(user, contents):
    task = Task()
    # agregar contenidos a Task, omitido por simplicidad
    zanzibar.write(user, "owner", task)

def change_task(user, task, new_contents):
    if zanzibar.check(user, "owner", task):
        task.change(new_contents)
    else:
        raise AuthorizationError
```

La implementación en Zanzibar es la más lejana a lo que eran las reglas SQL originales, pero Polar también resulta distinta en términos de cómo se usa. Debido a la complejidad de comparar soluciones de autorización con enfoques tan distintos, se comparará solo Pilpilang con Postgres en la mayoría de los casos de ejemplo.

### 4.1.3. Ventajas de Pilpilang

Corresponde preguntarse si se obtiene alguna ventaja al usar Pilpilang en vez de SQL para definir estos permisos. Para este ejemplo, no se observa ninguna ventaja[1] de Pilpilang por sobre SQL, incluso midiendo el programa más líneas de código que el original, y siendo igual de repetitivo[2].

Con respecto a las líneas de código, aunque no son la mejor métrica para comparar lenguajes, sí expresan para cada uno la relación entre contenido y ruido. En este caso, el ruido de Pilpilang se podría reducir "aprovechando" el hecho de que para todas las aplicaciones Supabase, los usuarios siempre se guardan en la misma tabla y con la misma estructura de manera estándar. Si se omite el bloque definición de actor, el ejemplo Pilpilang se reduce a 10 líneas.

Ahora se probará con un caso más complejo, para ver si en él, se aprecian más ventajas o desventajas de Pilpilang.

## 4.2. Gestor de Perfiles de Usuarios

Este ejemplo corresponde a una pequeña parte de muchas aplicaciones modernas multi-usuario: la edición de un perfil para cada usuario, con sus detalles asociados. Los perfiles son públicos, pero editarlos solo está permitido para sus usuarios correspondientes. Se pueden

[1] Una ventaja pequeña es que, gracias al type-checker de Pilpilang, es imposible equivocarse en una de las definiciones usando una llave foránea incorrecta.

[2] Aunque podría no ser tan repetitivo: existe una forma más eficiente de definir permisos con definiciones iguales en Pilpilang. Pero se debe mencionar que también existe esa posibilidad en SQL, por lo que se decidió hacer una traducción directa para mantener la justicia en la comparación.

asociar los perfiles con fotos, llamadas “avatars”, que también son públicas pero solo sus dueños las pueden evitar. Estas son las reglas que se definen a continuación:

```
-- "profiles" is a table for public profiles
alter table profiles
  enable row level security;

create policy "Public profiles are viewable by everyone." on profiles
  for select using (true);

create policy "Users can insert their own profile." on profiles
  for insert with check (auth.uid() = id);

create policy "Users can update own profile." on profiles
  for update using (auth.uid() = id);

-- Set up access controls for storage.
create policy "Avatar images are publicly accessible." on storage.objects
  for select using (bucket_id = 'avatars');

create policy "Anyone can upload an avatar." on storage.objects
  for insert with check (bucket_id = 'avatars');

create policy "Anyone can update their own avatar." on storage.objects
  for update using ( auth.uid() = owner )
  with check (bucket_id = 'avatars');
```

### 4.2.1. Equivalente Pilpilang

Su semi-equivalente en Pilpilang es el siguiente:

```
resource Profile {
    table "profiles"
    columns [
        user: User(id)
    ]
    keys ["id"]
}

actor User {
    table "auth.users"
    keys "id"
}

resource Object {
```

```
    table "storage.objects"
    columns: {
        bucket_id: String,
        owner: User(owner),
    }
}

can_select(u: User, p: Profile) if 1 = 1

can_insert(u: User, p: Profile) if p.user = u
can_update(u: User, p: Profile) if p.user = u

is_avatar(o: Object) if o.bucket_id = "avatars"

can_select(u: User, o: Object) if is_avatar(o)
can_insert(u: User, o: Object) if is_avatar(o)

can_update(u: User, o: Object) if u = o.owner && is_avatar(o)
```

En este ejemplo se pueden observar ciertas debilidades de la aplicabilidad de Pilpilang en comparación con el uso directo de SQL, las cuales se enumeran a continuación:

1. El recurso `Profile` tiene una columna `id` que es simultáneamente su llave primaria y una llave foránea a `User`. Es posible lidiar con esta situación definiendo una columna `user` que use el mismo `id` como llave foránea, pero esto es un *hack*, el manejo de estas situaciones no fue considerado originalmente como un requisito de diseño de Pilpilang.
2. El último `POLICY` del ejemplo en SQL ejecuta dos comprobaciones distintas en dos momentos distintos, mientras que su traducción a Pilpilang ejecuta ambas en ambos momentos. Esto significa que el comportamiento de ambas no es idéntico[3], mostrando una debilidad en la expresabilidad del lenguaje.

Aunque a grandes rasgos la traducción es buena, no es perfecta. En particular el punto 2 muestra un problema importante con el lenguaje tal como está implementado actualmente. Se podría solucionar agregando un nuevo permiso, más granular, que permitiera configurar la condición WITH CHECK de un `can_update`. Por ejemplo, `can_update_if_check`. Luego, el permiso tendría que dividirse en dos distintos, uno `can_update` para definir la condición que se ejecuta antes y después del cambio, y otro `can_update_if_check` que se ejecute solo después. Queda como propuesta a futuro.

Otra sugerencia para aumentar la usabilidad del lenguaje, inspirada en el *pattern matching* de [12], es la siguiente: Sería ideal el uso de tipos con cláusulas guarda, creando por ejemplo un

[3]En particular, la acción que un usuario tiene autorizado en la versión original pero no la de Pilpilang es la siguiente: Hacer una operación UPDATE a un objeto del que sea dueño pero que no sea un avatar, y convertirlo en su avatar. Esto se debe a que la condición WITH CHECK se evalúa solo después de realizar el UPDATE.

tipo `Avatar` de modo que, en lugar de escribir los permisos con la condición `is_avatar(o)`, esa condicion se agregara automáticamente al usar el tipo `Avatar` en la cabecera. Así que podría abstraer el hecho de que los avatares de la aplicación se implementan como objetos del esquema `storage`. Esta opción también se podrá analizar a más profundidad en trabajos futuros.

### 4.2.2. Equivalentes en lenguajes del estado del arte

Consideremos ahora las formas de expresar estas reglas en otros lenguajes de autorización.

#### Polar

En este ejemplo se omitirá el fragmento de Python que muestra la API para el uso de Polar. Se puede ver en el ejemplo anterior, y no hay mayor diferencia. Lo que sí se presenta es el archivo de reglas de Polar. Este archivo será nuevamente similar a la solución en Pilpilang, debido a la influencia de Polar en este último.

```
resource Profile {
    relations = {
        user: User
    };
}

actor User {
}

resource Avatar {
    relations = {
        owner: User,
    };
}

allow(u: User, "view", p: Profile);

allow(u: User, action, p: Profile) if
    p.user = u && action in ["insert", "update"];

allow(u: User, action, a: Avatar) if action in ["view", "create"];

allow(u: User, "modify", a: Avatar) if u = o.owner;
```

La solución es similar a Pilpilang, pero con el uso de un tipo específico `Avatar` en lugar de usar `Object`. Polar no está conectado con la base de datos, por lo que los tipos de sus

entidades no necesariamente dependen de la tabla en la que se guardan. Entonces, aunque estén en la tabla `storage.objects`, los avatares pueden ser de otro tipo.

#### Zanzibar

En esta ocasión, se pueden aprovechar las capacidades de Zanzibar más que en el ejemplo anterior. La siguiente configuración hace que el hecho de poder modificar un perfil permita también modificar su foto Avatar asociada.

```
name: "avatar"

relation {
 name: "editor"
 userset_rewrite {
  union {
   child { _this {} }
   child { tuple_to_userset {
     tupleset { relation: "owner" }
     computed_userset {
        object: $TUPLE_USERSET_OBJECT
        relation: "editor" }}}
}}}
```

En pocas palabras, esta configuración hace que cada editor de un perfil de usuario sea también editor de su avatar correspondiente. Sin embargo, sigue siendo necesario crear filas en la base de datos interna de Zanzibar desde el *Back-end* de la aplicación, para asociar cada perfil con su avatar. Esto se debe hacer aunque la información ya se encuentre guardada en la base de datos de la aplicación.

## 4.3. Clon de Slack

Slack es una aplicación de chat orientada a ambientes laborales, que tiene funcionalidad como administración de permisos y control de acceso mediante roles (RBAC)[4] para dar privilegios a distintos usuarios, de modo de permitir el uso de distintos canales de texto (chats). Es una buena aplicación para evaluar las capacidades de un sistema de autorización.

A continuación se presentan las reglas de autorización de un clon de Slack, simplificado para un solo servidor, en SQL. Ya que el ejemplo completo es muy grande, las reglas simples,

[4]El control de acceso mediante roles (RBAC por sus siglas en inglés) es un patrón de autorización común en muchas aplicaciones complejas. Tiene el beneficio de que entrega flexibilidad al problema de la autorización, ya que la decisión de si alguien está autorizado para algo depende de los roles que tenga, que se pueden modificar a lo largo del ciclo de vida de la aplicación. Este mismo ejemplo sirve para entender mejor estos beneficios.

datos dummy y definiciones no relacionadas con autorización se han omitido. El ejemplo se puede encontrar completo en el Anexo, en A.3. Este fragmento solo considera una parte interesante del ejemplo, que usa RBAC.

Considérese que los tipos `app_permission` y `app_role` son `enums` definidos por la aplicación, representados por strings. La tabla `channels` representa los canales (es decir chats de grupo). `messages` contiene los mensajes, cada uno de los cuales tiene una referencia a un usuario y un canal. `user_roles` da roles a los usuarios, describiendo si son moderadores, administradores o solo miembros. `role_permissions` describe los permisos asociados a un rol dado, como se puede ver al principio del fragmento.

```
insert into public.role_permissions (role, permission)
values
    ('admin', 'channels.delete'),
    ('admin', 'messages.delete'),
    ('moderator', 'messages.delete');

-- authorize with role-based access control (RBAC)
create function public.authorize(
  requested_permission app_permission,
  user_id uuid
)
returns boolean as $$
declare
  bind_permissions int;
begin
  select count(*)
  from public.role_permissions
  inner join public.user_roles on role_permissions.role = user_roles.role
  where role_permissions.permission = authorize.requested_permission
    and user_roles.user_id = authorize.user_id
  into bind_permissions;

  return bind_permissions > 0;
end;
$$ language plpgsql security definer;

create policy "Allow authorized delete access" on public.channels
    for delete using ( authorize('channels.delete', auth.uid()) );
create policy "Allow authorized delete access" on public.messages
    for delete using ( authorize('messages.delete', auth.uid()) );

-- policies simples, como los de los ejemplos anteriores
create policy "Allow individual insert access" on public.messages
    for insert with check ( auth.uid() = user_id );
create policy "Allow individual update access" on public.messages
    for update using ( auth.uid() = user_id );
```

```
-- ...
```

Para entender lo que se está haciendo en el ejemplo, se debe comprender el propósito de la funcion `public.authorize`. Esta función involucra a tres entidades: Los usuarios, las asociaciones usuario-rol y las asociaciones rol-permiso. Con estas entidades, determina si cada usuario tiene autorización para un permiso dado. El beneficio de este enfoque es que permite agregar roles o modificar sus asociaciones con permisos, sin que sea necesario modificar el código SQL, o cambiar las POLICYs. Solo se necesita agregar filas a la tabla `role_permissions`. Pero si se quiere agregar permisos nuevos, sí se debe cambiar las POLICYs, ya que se tendría que agregar más declaraciones CREATE POLICY iguales que las dos que ya existen.

### 4.3.1. Equivalente Pilpilang

```
actor User {
    ...
}

actor UserRole {
    table "public.user_roles"
    columns [
        role: String
    ]
    keys ["user_id"]
}

resource RolePerm {
    table "public.role_permissions"
    columns [
        role: String,
        permission: String
    ]
    keys ["role", "permission"]
}

resource Channel {
    table "public.channels"
    keys ["id"]
}

resource Message {
    table "public.messages"
    columns [
        sent_by: User(user_id),
    ]
```

```
    keys ["id"]
}

can_delete(u: UserRole, ch: Channel)[perm: RolePerm]
    if u.role = perm.role && perm.permission = "channels.delete"
can_delete(u: UserRole, m: Message)[perm: RolePerm]
    if u.role = perm.role && perm.permission = "messages.delete"

# permisos simples, como los de los ejemplos anteriores
can_insert(u: User, m: Message) if u = m.sent_by
can_update(u: User, m: Message) if u = m.sent_by
# ...
```

En este ejemplo se usa una funcionalidad de Pilpilang que puede resultar poco intuitiva: La definición de un `actor` que no es simplemente el `User` de siempre. De hecho, en la definición de `UserRole` se especifica que su llave es `user_id`, lo cual no es la llave primaria de la tabla en SQL. Es, sin embargo, este el propósito de permitir más de una declaración `actor` en cada programa Pilpilang: que muchas veces la base de datos podrá tener tablas que describen indirectamente reglas de autorización, en este caso `public.user_roles`, y resulta cómodo definir los permisos utilizando esas tablas directamente.

#### Usabilidad con RBAC

Se puede observar que el patrón RBAC es más simple de implementar en Pilpilang que en SQL, ya que no es necesario definir una función `authorize` que tome la decisión de autorización. Esto se debe a que la semántica del cuantificador existencial de Pilpilang ya consiste en una búsqueda de existencia entre todas las filas, exactamente lo que se realiza en la función `authorize`. Por lo tanto, se puede concluir que para este ejemplo (de mayor complejidad) Pilpilang sigue teniendo suficiente expresividad. Incluso resulta más eficiente que SQL en el sentido de que no es necesario definir una función extra[5].

Sin embargo, uno podría argumentar que el funcionamiento de la versión en Pilpilang depende del aprovechamiento de semánticas específicas, por lo que es menos flexible para las necesidades de cada aplicación, y requiere confianza del programador/programadora en sus conocimientos de los detalles del lenguaje.

### 4.3.2. Implementación en Polar y Zanzibar

El patrón RBAC es directo de implementar en ambos sistemas del estado del arte. Polar tiene funcionalidad específica para usar roles, lo cual se enfatiza en sus tutoriales sobre patrones de autorización. En Zanzibar una arquitectura común es usar roles como las aristas del grafo de autorización.

[5]En defensa de SQL, la función `authorize` no es estrictamente necesaria en ese lenguaje tampoco, pero las definiciones que no la usen resultan incómodamente largas, como las que emite Pilpilang al compilarse.

### Polar

En el lenguaje Polar, los recursos se pueden asociar con sus roles posibles, y usarlos para determinar resultados de autorización. Esta funcionalidad se utiliza en la siguiente implementación:

```
resource Channel {
  permissions = ["delete"];
  roles = ["moderator", "admin"];

  # Un admin puede eliminar Channels
  "delete" if "admin";

  # Un admin tiene todos los permisos de un moderator
  "moderator" if "admin";
 }

resource Message {
  permissions = ["delete"];
  relations = {channel: Channel};

  # Un moderator de un Channel puede eliminar sus Messages
  "delete" if "moderator" on "channel";
}
```

Se puede ver que la solución es muy corta, y con los comentarios resulta también fácil de entender. Sin embargo, no es completamente equivalente a la solución que se vio previamente, ya que no es posible agregar o quitar permisos a ciertos roles dinámicamente. A diferencia de las soluciones de Postgres o Pilpilang, no se está usando una tabla `role_permissions` para asociar los roles a los permisos, sino que estas asociaciones están *fijas* en la configuración.

### Zanzibar

Uno de los principios de Zanzibar es tener una base de datos interna para la autorización. Esta decisión resulta acertada en este ejemplo, ya que las tablas `user_roles` y `role_permissions` son solo para autorizar, se podrían mover completamente a la base de datos de Zanzibar sin crear redundancia al hacerlo.

Los contenidos de la tabla `user_roles` se expresarían mediante tuplas como las siguientes:[6]

```
channel:13#admin@5
channel:13#moderator@7
```

[6]Para entender esta notación en más detalle, referirse a [20], sección 2.1

Donde 5 y 7 son IDs numéricos de usuarios, y 13 es el ID de un canal de chat. También se debe traducir la tabla `role_permissions`, pero al hacerlo se observará un problema.

```
channel:13#deleter@channel:13#admin
channel:14#deleter@channel:14#admin
message:523#deleter@channel:14#moderator
message:258#deleter@channel:13#moderator
...
```

Puede parecer absurdo tener que guardar en base de datos las acciones posibles para cada canal y mensaje, pero a cambio se obtiene mucha flexibilidad. Una alternativa es referirse a los roles directamente desde el *back-end*, es decir, no usar el rol "deleter" sino simplemente "moderator". Otra alternativa es usar una configuración compleja para automatizar el rol "deleter", así como se hizo en el ejemplo con Polar. Esa configuración se puede encontrar en el Anexo B.

## 4.4. Conclusión de la Evaluación

Observando los tres ejemplos, se pueden concluir algunas ideas. En primer lugar, la aplicabilidad de Pilpilang como lenguaje de autorización, comparada con la de SQL, resulta positivamente evaluada, ya que los tres ejemplos en SQL se pudieron reescribir en Pilpilang. El segundo ejemplo tiene una pequeña diferencia de su comportamiento, pero si se quisiera tener exactamente el mismo, agregar esa funcionalidad a Pilpilang no sería difícil, bastaría con agregar un permiso. Así que se puede concluir que la aplicabilidad es buena. Eso sí, el tamaño muestral de ejemplos es muy bajo (3)[7], por lo que la conclusión es una afirmación de baja confianza.

En segundo lugar, está la eficiencia en términos de líneas de código. En esto, es de esperar normalmente que un lenguaje de dominio específico (DSL) sea más eficiente que uno de propósito más general, pero esto no se observa en los ejemplos. La definición de entidades en Pilpilang (actores y recursos) ocupa mucho espacio[8], más del que se gana escribiendo los permisos más compactamente. Sin embargo, el tercer ejemplo muestra que en casos de mayor complejidad, el DSL mantiene un tamaño pequeño y legible, a diferencia de SQL que crece significativamente con la creación de una función para la autorización. Los lenguajes del estado del arte resultan difíciles de comparar, ya que se usan de maneras distintas, por lo que no es posible hacer programas equivalentes a los de SQL.

Por último, se observan varias oportunidades de mejora para Pilpilang. En la sección 4.2, de gestión de usuarios, se mencionan dos específicas: Una es agregar un permiso nuevo,

[7]De hecho, como autor de Pilpilang, se me ocurre un patrón que no sería expresable en el lenguaje: Una aplicación similar a Dropbox, donde tener acceso a una carpeta automáticamente da acceso a las que están contenidas en ella, y así recursivamente. Esto se puede expresar en SQL usando `WITH RECURSIVE`, pero no en Pilpilang. Simplemente fue suerte que ninguno de los ejemplos usara un patrón similar.

[8]Esto se podría mejorar a futuro, incluyendo una optimización que obtenga la información de cada entidad desde la base de datos.

variante de `can_update`. La otra es que se puedan crear entidades a partir de otras, ejecutando condiciones automáticamente al usarlas.

# Capítulo 5

# Conclusión

En este trabajo, se diseñó e implementó un lenguaje de dominio específico para la definición de reglas y permisos de autorización. La implementación se trató de un transpilador que transformara estos permisos a SQL, de modo que se pudieran introducir a bases de datos PostgreSQL, para luego aplicar estas reglas en cada consulta.

El lenguaje se diseñó en gran parte basado en lenguajes más avanzados de autorización. En particular, se tomaron patrones de diseño de Polar [12] y Zanzibar [20], adaptándolos dentro de lo posible al problema específico de aplicar las reglas en bases de datos, y aprovechando las oportunidades dadas por ella. El lenguaje se basa en las relaciones existentes en BBDD para regular el acceso entre actores y recursos.

Respecto al transpilador, éste se implementó en Haskell e incluye las etapas de *parsing*, revisión de tipos y emisión de código como resultado. De este modo, mejora el nivel de confianza en la traducción generada, asegurándose de que cada operación sea válida para los tipos involucrados. Además, la revisión de tipos ayuda a la ergonomía del lenguaje, ya que permite hacer cosas como la inserción automática de llaves primarias en comparaciones.

## 5.1. Objetivos alcanzados y no alcanzados

En la sección 1.2, se describieron los objetivos que se pretendía lograr con este trabajo. En primer lugar, se deseaba diseñar e implementar el lenguaje Pilpilang, con patrones de lenguajes de autorización del estado del arte. Este objetivo fue mayormente cumplido, ya que Pilpilang fue diseñado a partir de patrones del estado del arte, una implementación funcional fue realizada, y los resultados de la evaluación fueron mayormente positivos.

Además del objetivo general, se decidieron varios objetivos específicos al proponer este trabajo. Estos objetivos son, en su mayoría, las partes de escribir un compilador con checkeo de tipos. Como se creó la implementación de Pilpilang, esos objetivos específicos se cumplieron.

Un objetivo específico no fue logrado: la obtención de la estructura de la base de datos,

para minimizar la cantidad de información sobre ella que se debe incluir en cada programa Pilpilang. Durante la realización del proyecto, se decidió que esta funcionalidad sería opcional, lo cual minimizó su prioridad y derivó en que no se implementara. En el capítulo 4 de Evaluación, se menciona que la detección automática de columnas y llaves de entidades queda como trabajo futuro que podría mejorar la usabilidad del lenguaje.

En conclusión, aunque no se alcanzaron todos los objetivos decididos inicialmente, sí se logró la gran mayoría, los esenciales para alcanzar el objetivo general.

## 5.2. Consecuencias del trabajo

La evaluación reveló que, aunque el lenguaje no haya tenido tanta funcionalidad como se deseaba, sí resultaba aplicable a los problemas que enfrentó. También reveló ventajas que tenía por sobre el uso directo de SQL, gracias a su diseño más acorde a las buenas prácticas de autorización. Por lo tanto, resulta exitoso en demostrar la factibilidad del uso de un DSL de autorización para bases de datos relacionales.

Sin embargo, no es difícil idear obstáculos al uso de Pilpilang en el mundo real, limitando el impacto que pueda tener más allá de prueba de concepto. La seguridad de los datos es un tema sensible y esencial, que amerita el uso de proyectos con el apoyo de empresas o comunidades establecidas, con un *roadmap* definido. Los lenguajes además suelen tener herramientas relacionadas, como un depurador, que Pilpilang no tiene. En resumen, falta trabajo y la promesa de mantención para que Pilpilang se utilice.

## 5.3. Trabajo Futuro

A lo largo de este documento, y sobre todo en la Evaluación, se han mencionado ideas de trabajo futuro en el tema. En esta sección se recopilan, y se añaden algunas más.

- Se podría obtener información de la estructura de la base de datos, usando las herramientas de introspección de PostgreSQL, para mejorar la usabilidad de Pilpilang. Los tipos de las columnas o quizá sus descripciones completas podrían omitirse con funcionalidad así, simplificando los bloques de declaración de recursos y actores.
- En ocasiones, una misma tabla en una base de datos tendrá múltiples tipos de actores o recursos. Para mejorar la ergonomía del DSL, se podría agregar la posibilidad de crear entidades especiales a partir de filtros sobre entidades existentes. Esto permitiría, por ejemplo, crear actores distintos para usuarios normales, fiscalizadores y administradores, con tipos distintos, aún cuando los tres se guarden en la misma tabla.
- Algún tipo de introspección a las reglas que se están creando. Como usuario, aprecio cuando un lenguaje me permite obtener información sobre por qué se está obteniendo un resultado en lugar de otro.

- Las bases de datos se pueden usar de muchas maneras. La autorización mediante Row-Level-Security de Postgres se usa frecuentemente para exponer bases de datos a la red sin un *back-end* [19, 14], pero no tiene por qué estar limitada a eso. Quizá se podría trabajar en integrar algún *framework* como Django con este mecanismo de autorización. Ya se han creado paquetes de permisos para Django usando otras estrategias, que pueden servir de inspiración [18].

La lista dada contiene ideas de dificultades variables y utilidades variables. No todas están estrictamente relacionadas con Pilpilang, pero sí están relacionadas con el campo de la autorización en bases de datos relacionales.

# Bibliografía


[1] Oracle and/or its affiliates. *MySQL: Documentation: Access Control and Account Management*. 2022. URL: https://dev.mysql.com/doc/refman/8.0/en/access-control.html (visitado 11-12-2022).

[2] Oracle and/or its affiliates. *Oracle SQL: Managing Access Control*. 2014. URL: https://docs.oracle.com/database/121/TTOPR/accesscontrol.htm#TTOPR236 (visitado 11-12-2022).

[3] FIDO Alliance. *FIDO2: Web Authentication (WebAuthn)*. URL: https://fidoalliance.org/fido2-2/fido2-web-authentication-webauthn/ (visitado 02-07-2022).

[4] Edwin Brady. "IDRIS —: systems programming meets full dependent types". En: *Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011*. Ed. por Ranjit Jhala y Wouter Swierstra. ACM, 2011. ISBN: 978-1-4503-0487-0. DOI: 10.1145/1929529.1929536. URL: http://doi.acm.org/10.1145/1929529.1929536.

[5] Lars M. Garshol. *BNF and EBNF: What are they and how do they work?* 2008. URL: https://www.garshol.priv.no/download/text/bnf.html (visitado 11-12-2022).

[6] Andy Gill y Simon Marlow. *Happy: A parser generator system for Haskell*. 2022. URL: https://haskell-happy.readthedocs.io/en/latest/introduction.html (visitado 11-12-2022).

[7] G. Gonzales y contributors. *The Dhall Configuration Language*. 2017. URL: https://dhall-lang.org/ (visitado 02-07-2022).

[8] The PostgreSQL Global Development Group. *Documentation: CREATE POLICY*. 2021. URL: https://www.postgresql.org/docs/14/sql-createpolicy.html (visitado 11-12-2022).

[9] The PostgreSQL Global Development Group. *Documentation: Row Security Policies*. 2021. URL: https://www.postgresql.org/docs/14/ddl-rowsecurity.html (visitado 11-12-2022).

[10] The PostgreSQL Global Development Group. *PostgreSQL Release 9.5*. 2016. URL: https://www.postgresql.org/docs/9.5/release-9-5.html (visitado 11-12-2022).

[11] Authzed Inc. *SpiceDB and Authzed*. 2021. URL: https://docs.authzed.com (visitado 02-07-2022).

[12] Oso Security Inc. *Polar Language Reference*. 2021. URL: https://docs.osohq.com/reference/polar.html (visitado 02-07-2022).

[13] Supabase Inc. *GoTrue: A JWT based API for managing users and issuing JWT tokens*. 2021. URL: https://github.com/supabase/gotrue/ (visitado 02-07-2022).

[14] Supabase Inc. *Supabase, the Open Source Firebase Alternative*. 2021. URL: https://supabase.com (visitado 02-07-2022).

[15] M. Jones, J. Bradley y N.Sakimura. *JSON Web Token (JWT)*. RFC 7519. RFC Editor, mayo de 2015. URL: https://www.rfc-editor.org/rfc/rfc4180.txt.

[16] M. Karpov y contributors. *Megaparsec: Industrial-strength monadic parser combinator library*. 2014. URL: https://github.com/mrkkrp/megaparsec (visitado 02-07-2022).

[17] Daan Leijen y Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Inf. téc. UU-CS-2001-27. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007. Jul. de 2001. URL: https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/.

[18] Fabio C. Barrionuevo da Luz y Jeff Triplett. *Django Packages: Permissions*. 2022. URL: https://djangopackages.org/grids/g/perms/ (visitado 11-12-2022).

[19] J. Nelson y S. Chavez. *PostgREST*. 2017. URL: https://postgrest.org/en/stable (visitado 02-07-2022).

[20] Ruoming Pang et al. "Zanzibar: Google's Consistent, Global Authorization System". En: *2019 USENIX Annual Technical Conference (USENIX ATC '19)*. Renton, WA, 2019.

# Anexos

# Anexo A

# Ejemplos completos de la Evaluación

En Evaluación, se presentan ejemplos de reglas de autorización, para distintos casos de uso, creados por Supabase. Sin embargo, se presentan reducidos a sus partes más relevantes para el presente trabajo. Aquí se presentan completos:

## A.1. Lista de Tareas Pendientes

```
create table todos (
  id bigint generated by default as identity primary key,
  user_id uuid references auth.users not null,
  task text check (char_length(task) > 3),
  is_complete boolean default false,
  inserted_at timestamp with time zone
    default timezone('utc'::text, now()) not null
);

alter table todos enable row level security;

create policy "Individuals can create todos." on todos for
    insert with check (auth.uid() = user_id);

create policy "Individuals can view their own todos. " on todos for
```

```
    select using (auth.uid() = user_id);

create policy "Individuals can update their own todos." on todos for
    update using (auth.uid() = user_id);

create policy "Individuals can delete their own todos." on todos for
    delete using (auth.uid() = user_id);
```

## A.2. Gestión de Usuarios

```
-- Create a table for public profiles
create table profiles (
  id uuid references auth.users
    not null primary key,
  updated_at timestamp with time zone,
  username text unique,
  full_name text,
  avatar_url text,
  website text,

  constraint username_length check (char_length(username) >= 3)
);
-- Set up Row Level Security (RLS)
-- See https://supabase.com/docs/guides/auth/row-level-security
    -- for more details.
alter table profiles
  enable row level security;

create policy "Public profiles are viewable by everyone." on profiles
  for select using (true);

create policy "Users can insert their own profile." on profiles
  for insert with check (auth.uid() = id);

create policy "Users can update own profile." on profiles
  for update using (auth.uid() = id);

-- This trigger automatically creates a profile entry
-- when a new user signs up via Supabase Auth.
-- See https://supabase.com/docs/guides/auth/managing-user-data
-- for more details.
create function public.handle_new_user()
returns trigger as $$
begin
  insert into public.profiles (id, full_name, avatar_url)
```

```
  values (new.id, new.raw_user_meta_data->>'full_name',
    new.raw_user_meta_data->>'avatar_url');
  return new;
end;
$$ language plpgsql security definer;
create trigger on_auth_user_created
  after insert on auth.users
  for each row execute procedure public.handle_new_user();

-- Set up Storage!
insert into storage.buckets (id, name)
  values ('avatars', 'avatars');

-- Set up access controls for storage.
-- See https://supabase.com/docs/guides/storage#policy-examples
    -- for more details.
create policy "Avatar images are publicly accessible." on storage.objects
  for select using (bucket_id = 'avatars');

create policy "Anyone can upload an avatar." on storage.objects
  for insert with check (bucket_id = 'avatars');

create policy "Anyone can update their own avatar." on storage.objects
  for update using ( auth.uid() = owner )
  with check (bucket_id = 'avatars');
```

## A.3. Clon de Slack

```
--
-- For use with https://github.com/supabase/supabase/tree/master/
   examples/slack-clone/nextjs-slack-clone
--

-- Custom types
create type public.app_permission as enum ('channels.delete', 'messages
   .delete');
create type public.app_role as enum ('admin', 'moderator');
create type public.user_status as enum ('ONLINE', 'OFFLINE');

-- USERS
create table public.users (
  id          uuid not null primary key, -- UUID from auth.users
  username    text,
  status      user_status default 'OFFLINE'::public.user_status
);
comment on table public.users is 'Profile data for each user.';
```

```
comment on column public.users.id is 'References the internal Supabase
    Auth user.';

-- CHANNELS
create table public.channels (
  id            bigint generated by default as identity primary key,
  inserted_at   timestamp with time zone default timezone('utc'::text,
      now()) not null,
  slug          text not null unique,
  created_by    uuid references public.users not null
);
comment on table public.channels is 'Topics and groups.';

-- MESSAGES
create table public.messages (
  id            bigint generated by default as identity primary key,
  inserted_at   timestamp with time zone default timezone('utc'::text,
      now()) not null,
  message       text,
  user_id       uuid references public.users not null,
  channel_id    bigint references public.channels on delete cascade not
      null
);
comment on table public.messages is 'Individual messages sent by each
    user.';

-- USER ROLES
create table public.user_roles (
  id        bigint generated by default as identity primary key,
  user_id   uuid references public.users on delete cascade not null,
  role      app_role not null,
  unique (user_id, role)
);
comment on table public.user_roles is 'Application roles for each user.
    ';

-- ROLE PERMISSIONS
create table public.role_permissions (
  id            bigint generated by default as identity primary key,
  role          app_role not null,
  permission    app_permission not null,
  unique (role, permission)
);
comment on table public.role_permissions is 'Application permissions
    for each role.';

-- authorize with role-based access control (RBAC)
create function public.authorize(
  requested_permission app_permission,
  user_id uuid
```

```
)
returns boolean as $$
declare
  bind_permissions int;
begin
  select count(*)
  from public.role_permissions
  inner join public.user_roles on role_permissions.role = user_roles.
     role
  where role_permissions.permission = authorize.requested_permission
    and user_roles.user_id = authorize.user_id
  into bind_permissions;

  return bind_permissions > 0;
end;
$$ language plpgsql security definer;

-- Secure the tables
alter table public.users enable row level security;
alter table public.channels enable row level security;
alter table public.messages enable row level security;
alter table public.user_roles enable row level security;
alter table public.role_permissions enable row level security;
create policy "Allow logged-in read access" on public.users for select
   using ( auth.role() = 'authenticated' );
create policy "Allow individual insert access" on public.users for
   insert with check ( auth.uid() = id );
create policy "Allow individual update access" on public.users for
   update using ( auth.uid() = id );
create policy "Allow logged-in read access" on public.channels for
   select using ( auth.role() = 'authenticated' );
create policy "Allow individual insert access" on public.channels for
   insert with check ( auth.uid() = created_by );
create policy "Allow individual delete access" on public.channels for
   delete using ( auth.uid() = created_by );
create policy "Allow authorized delete access" on public.channels for
   delete using ( authorize('channels.delete', auth.uid()) );
create policy "Allow logged-in read access" on public.messages for
   select using ( auth.role() = 'authenticated' );
create policy "Allow individual insert access" on public.messages for
   insert with check ( auth.uid() = user_id );
create policy "Allow individual update access" on public.messages for
   update using ( auth.uid() = user_id );
create policy "Allow individual delete access" on public.messages for
   delete using ( auth.uid() = user_id );
create policy "Allow authorized delete access" on public.messages for
   delete using ( authorize('messages.delete', auth.uid()) );
create policy "Allow individual read access" on public.user_roles for
   select using ( auth.uid() = user_id );
```

```
-- Send "previous data" on change
alter table public.users replica identity full;
alter table public.channels replica identity full;
alter table public.messages replica identity full;

-- inserts a row into public.users and assigns roles
create function public.handle_new_user()
returns trigger as $$
declare is_admin boolean;
begin
  insert into public.users (id, username)
  values (new.id, new.email);

  select count(*) = 1 from auth.users into is_admin;

  if position('+supaadmin@' in new.email) > 0 then
    insert into public.user_roles (user_id, role) values (new.id, '
       admin');
  elsif position('+supamod@' in new.email) > 0 then
    insert into public.user_roles (user_id, role) values (new.id, '
       moderator');
  end if;

  return new;
end;
$$ language plpgsql security definer;

-- trigger the function every time a user is created
create trigger on_auth_user_created
  after insert on auth.users
  for each row execute procedure public.handle_new_user();

/**
 * REALTIME SUBSCRIPTIONS
 * Only allow realtime listening on public tables.
 */

begin;
  -- remove the realtime publication
  drop publication if exists supabase_realtime;

  -- re-create the publication but don't enable it for any tables
  create publication supabase_realtime;
commit;

-- add tables to the publication
alter publication supabase_realtime add table public.channels;
alter publication supabase_realtime add table public.messages;
alter publication supabase_realtime add table public.users;
```

```
-- DUMMY DATA
insert into public.users (id, username)
values
    ('8d0fd2b3-9ca7-4d9e-a95f-9e13dded323e', 'supabot');

insert into public.channels (slug, created_by)
values
    ('public', '8d0fd2b3-9ca7-4d9e-a95f-9e13dded323e'),
    ('random', '8d0fd2b3-9ca7-4d9e-a95f-9e13dded323e');

insert into public.messages (message, channel_id, user_id)
values
    ('Hello World', 1, '8d0fd2b3-9ca7-4d9e-a95f-9e13dded323e'),
    ('Perfection is attained, not when there is nothing more to add,
       but when there is nothing left to take away.', 2, '8d0fd2b3-9ca7
       -4d9e-a95f-9e13dded323e');

insert into public.role_permissions (role, permission)
values
    ('admin', 'channels.delete'),
    ('admin', 'messages.delete'),
    ('moderator', 'messages.delete');
```

# Anexo B

# Implementación alternativa del ejemplo 3 en Zanzibar

La configuración para asociar permisos y roles se puede realizar de la siguiente manera para los canales:

```
name: "channel"

relation {
    name: "admin"
}

relation {
    name: "moderator"
    userset_rewrite {
        union {
        child { _this {} }
        child { computed_userset {
            relation: "admin" }}
}}}

relation {
    name: "deleter"
    userset_rewrite {
        child { computed_userset {
            relation: "admin" }}
}}}
```

y para los mensajes se debe usar otra configuración:

```
name: "message"
```

```
relation {
    name: "deleter"
    userset_rewrite {
        tuple_to_userset {
            tupleset { relation: "channel" }
            computed_userset {
                object: $TUPLE_USERSET_OBJECT
                relation: "moderator" }
}}}
```

La cual obtiene los moderadores del canal de cada mensaje y les asigna el rol “deleter” en ese mensaje.