Universidad de la República
Facultad de Ingeniería

Un sistema de tipos gradual para el lenguaje funcional Elixir

Proyecto de grado

Mauricio Cassola
Agustín Talagorría

Tutores

Alberto Pardo
Marcos Viera

URUGUAY, Montevideo Diciembre 2020

Agradecimientos

Agradecemos a todas las personas que nos dieron apoyo y aliento durante la elaboración de este proyecto. Entre ellas a nuestras familias, amigos y novias por la comprensión y aguante. A nuestras empresas por permitirnos llevar a cabo nuestras reuniones y brindarnos facilidades de estudio. También agradecemos a Marcos y Alberto, nuestros tutores, por la confianza para trabajar en un área conocida por ellos, pero en un lenguaje nuevo, y por la buena energía y disposición.

Resumen

Elixir es un lenguaje funcional, relativamente nuevo, con tipado dinámico, que busca ser muy potente y tener una sintaxis moderna. Además, es un lenguaje *open source*, lo que permite el estudio de su código fuente y la colaboración de la comunidad en su crecimiento.

En este proyecto se propone un sistema de tipos que haga posible ejecutar un chequeo de tipos de forma estática para un fragmento significativo de Elixir, sin perder su esencia y flexibilidad. El sistema de tipos que se presenta no requiere ningún tipo de cambios en la sintaxis del lenguaje. Está basado en el concepto de tipado gradual, donde el programador decide el grado de chequeos estáticos que quiere que se realicen en su programa. La información de los tipos es proporcionada mediante firmas de funciones con la notación que Elixir provee, a diferencia de otras soluciones donde las colisiones de tipos son detectadas por inferencia. Este enfoque busca combinar los beneficios que el tipado estático y dinámico ofrecen.

Se implementa un prototipo, también en Elixir, de un analizador de tipos basado en el sistema de tipos definido. Además, se realizan experimentos de uso de la biblioteca así como de aceptación del enfoque del sistema de tipos.

El trabajo deriva en un paper presentado en *SBLP 2020: 24th Brazilian Symposium on Programming Languages*.

Palabras clave: programación funcional, elixir, sistema de tipos, *gradual typing*.

Índice general

Capítulo 1

Introducción

Un paradigma de programación consiste en un método para llevar a cabo cómputos y la forma en la que deben estructurarse y organizarse las tareas que debe realizar un programa, es decir, se basa en un conjunto de reglas y principios de diseño. El paradigma funcional es uno de ellos, el cual es un paradigma de programación declarativa basado en el uso de funciones matemáticas. Por otro lado, los lenguajes de programación son una pieza fundamental en la ingeniería de software. Existe una gran variedad de ellos, basados en conceptos y fundamentos distintos, y con diferentes propósitos. Para aplicar un paradigma de programación es necesario un lenguaje que lo implemente. En cuanto al paradigma funcional, existen lenguajes fuertemente basados en él, como es el ejemplo de Haskell (Hudak, Hughes, Peyton Jones, y Wadler, 2007), pero también hay muchos lenguajes, que no son basados enteramente en el paradigma, pero han intentado implementarlo parcialmente, como lo es por ejemplo, Javascript (Jensen, Møller, y Thiemann, 2009). Es claro que este paradigma ha sido relacionado en mayor grado con la academia que con la industria, lo que puede ser debido a su fuerte relación con la matemática, en particular con el cálculo lambda. Sin embargo, últimamente la programación funcional ha tomado impulso para solucionar problemas como la concurrencia y la velocidad de procesamiento, especialmente en el área del desarrollo web. Esto conlleva a la creación de nuevos lenguajes y con diferentes características, entre los que se encuentra Elixir.

Elixir (Jurić, 2015) es un lenguaje de programación funcional de propósito general que se ejecuta en la máquina virtual de Erlang (BEAM) (Stenman, 2018). Se centra en la productividad del desarrollador, con un gran soporte para la concurrencia y la programación web ágil como su columna vertebral. Aprovecha todos los beneficios que ofrece la Erlang VM y su ecosistema, mientras que su sintaxis está influenciada por un lenguaje de programación moderno como lo es Ruby (Thomas, Fowler, y Hunt, 2013). También viene con un conjunto de herramientas que simplifica la gestión de proyectos, las pruebas, el empaquetado y la creación de documentación.

Como Erlang, Elixir es un lenguaje de tipado dinámico. Hay herramientas,

como Dialyxir (Dialyxir, 2020), que hacen posible realizar algún tipo de chequeo estático. Dialyxir simplemente se conecta con una herramienta de Erlang, llamada Dialyzer (Sagonas y Luna, 2008), que realiza un análisis estático en el BEAM bytecode, por lo que los mensajes no son de utilidad para el desarrollador, en particular, para saber dónde están los errores en el código original Elixir o cual es el motivo de su ocurrencia. Se basa en el concepto de *success typing* (Lindahl y Sagonas, 2006) para encontrar errores de tipo en el código. Similar a lo que es *soft typing* (Cartwright y Fagan, 2004), tiene la desventaja de ser bastante permisivo, mientras que es flexible y más adaptable al estilo de tipado dinámico.

El problema de tener tipado dinámico y estático para capitalizar sus fortalezas y remediar sus debilidades ha sido un tema de estudio durante años, y hay muchas soluciones propuestas. En este proyecto de grado se presenta un sistema de tipos para un fragmento de Elixir que permite realizar un chequeo de tipos estático sin perder la esencia y flexibilidad del lenguaje. Nuestro enfoque está inspirado en el llamado *gradual typing* (Castagna, Lanvin, Petrucciani, y Siek, 2019; Siek y Taha, 2007, 2006), que está presente en extensiones de muchos lenguajes dinámicos como TypeScript, PHP, Typed Racket, Typed Clojure, entre otros. El sistema de tipos propuesto se basa en subtipado y es compatible con código Elixir no tipado, ya que permite la presencia de fragmentos de código sin tipos. El programador decide qué partes del código deben verificarse de forma estática. La información de tipos se proporciona mediante firmas en las funciones que se declaran en términos de las directivas de especificación de tipos de Elixir (*@spec*). Una característica de nuestro sistema de tipos es que no produce ningún cambio en el código generado. Las partes del código que no podemos verificar de forma estática debido a falta de información de tipos, se verificarán en tiempo de ejecución.

Un producto derivado del presente trabajo es un artículo publicado en SBLP 2020 (Cassola, Talagorria, Pardo, y Viera, 2020).

El resto de este documento se encuentra dividido en cinco capítulos. El segundo capítulo se focaliza principalmente en detallar el marco teórico que se considera necesario. El capítulo tres, presenta el sistema de tipos que se propone. El capítulo cuatro, presenta la implementación que permite chequear código Elixir estáticamente, el análisis de resultados y las pruebas realizadas al lenguaje con la extensión y su aceptación en la industria. En el último capítulo se tienen las conclusiones de este proyecto, así como el trabajo a futuro.

Existe un glosario el cual se puede consultar en cualquier momento para obtener información de algunos términos poco usuales o que puedan ser desconocidos por el lector.

Capítulo 2

Marco Teórico

En este capítulo se introducen resumidamente los conceptos que se consideran necesarios para la comprensión del proyecto. En particular, se introduce a Elixir y su ecosistema así como los tipos en los lenguajes de programación y su importancia.

2.1. Elixir

Elixir (Jurić, 2015)(Thomas, 2014) es un lenguaje de propósito general basado en el paradigma funcional que se ejecuta en la máquina virtual de Erlang (BEAM) (Stenman, 2018). Su foco está en la productividad del desarrollador y cuenta con un gran soporte para la concurrencia y la programación web ágil.

Fue creado por José Valim en 2012, y su velocidad y capacidades lo han estado difundiendo en las industrias de telecomunicaciones, comercio electrónico y finanzas. Esto se debe en gran parte a que toma todas las ventajas que ofrece la máquina virtual de Erlang y su ecosistema. Para entender un poco más, a continuación se brinda una breve descripción de lo que se considera más relevante sobre Erlang y los beneficios que ofrece.

2.1.1. Ecosistema

Erlang es una plataforma de desarrollo para construir sistemas escalables y confiables, que proporcionen servicio constantemente, con poco o ningún tiempo de inactividad. En particular, Erlang brinda soporte para desafíos técnicos no funcionales, como lo es la concurrencia.

La concurrencia es el pilar de los sistemas Erlang. Casi todos los sistemas de producción no triviales basados en Erlang son altamente concurrentes, incluso el lenguaje de programación a veces se denomina lenguaje orientado a la concurrencia. En lugar de depender de hilos pesados y los procesos del sistema operativo, Erlang toma la simultaneidad en sus propias manos. La primitiva de concurrencia básica se denomina proceso Erlang (no debe confundirse con

proceso o subproceso del sistema operativo), y los sistemas típicos de Erlang ejecutan miles o incluso millones de estos procesos Erlang. La máquina virtual Erlang (BEAM), utiliza sus propios *schedulers* para distribuir la ejecución de procesos sobre los núcleos de CPU disponibles, paralelizando así la ejecución tanto como sea posible.

En particular, los códigos Elixir corren en estos procesos e intercambian información a través de mensajes. No es raro tener miles de procesos ejecutándose simultáneamente en la misma máquina, lo que permite que todos los recursos de la máquina se usen de la manera más eficiente posible (escalabilidad vertical). También, un proceso puede comunicarse con otros procesos que se ejecutan en diferentes máquinas en la misma red, lo que proporciona una base para la distribución, que puede ser aprovechada por los desarrolladores para coordinar el trabajo en múltiples nodos (escalabilidad horizontal). Para esto, Elixir proporciona supervisores que describen cómo reiniciar partes de un sistema cuando las cosas salen mal, volviendo a un estado inicial conocido que garantiza que funcione, lo que lo hace un lenguaje robusto, sumamente escalable y tolerante a fallas.

2.1.2. Tipado del lenguaje

Como Erlang, Elixir es un lenguaje con tipado dinámico. Existen herramientas o extensiones, como *Dialyxir* (Dialyxir, 2020), que hacen posible ejecutar cierto tipo de chequeo estático. Esta herramienta no es más que un pasaje a la herramienta de Erlang *Dialyzer* (Sagonas y Luna, 2008), la cual realiza un análisis en el código BEAM bytecode. Debido a esto, los errores en Dialyxir no son muy descriptivos en cuanto al lugar donde se produjo el error en el código Elixir o por qué sucedió.

Por otro lado, su tipado es implícito aunque no cuenta con un algoritmo de inferencia robusto y extensivo a todo el lenguaje.

Que sea de tipado dinámico e implícito tiene muchas ventajas para lograr un lenguaje flexible y ágil para el desarrollo web pero también muchas desventajas en las que este proyecto se inspira y está basado. El problema de combinar tipado dinámico y estático para capitalizar sus ventajas y remediar sus desventajas ha sido tema de estudio por muchos años. Hay muchas soluciones propuestas que se irán describiendo, entre las que destacaremos al tipado gradual (Siek y Taha, 2006, 2007).

Existen ya algunas propuestas basadas en este enfoque para Elixir, como lo es *Gradualixir*, pero al igual que *Dialyxir*, no es más que un pasaje a una herramienta de Erlang denominada *Gradualizer* (Gradualizer, 2020). Esta última se basa en un sistema de tipos gradual, que chequea estáticamente al programa a partir de las notaciones de tipos que se agregan.

2.1.3. Enfoque funcional

Elixir es un lenguaje con características funcionales.

Un lenguaje de programación funcional nos permite pensar en términos de funciones que transforman datos. Dichas transformaciones nunca mutan los datos, sino que por el contrario en cada aplicación de una función se crea potencialmente una nueva versión de los datos. En adición, este tipo de lenguajes promueve un estilo de codificación que ayuda a los desarrolladores a escribir código breve, conciso y fácil de mantener.

Entrando un poco más en detalle en algunos de los principios funcionales en los que se basa el lenguaje, se destaca que Elixir es esencialmente de **evaluación estricta** con características de **evaluación perezosa** mediante el módulo *Stream* que permite operar y componer funciones sobre conjuntos enumerables, como pueden ser listas o mapas. Además, tiene la característica de ser un lenguaje de alto orden que se usa de forma limitada (vía lambda abstracciones). Por último, se destaca el uso de **pattern-matching** para asociar patrones con valores.

Una herramienta interesante que provee el lenguaje es el **Property Based Testing**, inspirado fuertemente en QuickCheck para Haskell (Claessen y Hughes, 2011).

2.1.4. Desarrollo web

Aunque Elixir puede usarse como lenguaje de desarrollo para aplicaciones con distintos propósitos, el desarrollo web es sin lugar a dudas el uso más común y conocido. En ese sentido, se encuentra *Phoenix* (McCord, 2019), un framework creado especialmente para facilitar el desarrollo web con Elixir.

Es un framework de desarrollo web escrito en Elixir que implementa el patrón modelo, vista y controlador (MVC), del lado del servidor. Muchos de sus componentes y conceptos son familiares a otros frameworks web como Ruby on Rails, o Django de Python. Ofrece lo mejor de ambos mundos dado que brinda una alta productividad al desarrollador y un alto rendimiento en las aplicaciones, ya que por ejemplo, utiliza plantillas precompiladas para una gran velocidad, y además, proporciona comunicación en tiempo real a clientes externos a través de *web sockets* y/o *polling* utilizando su funcionalidad de *channels*.

2.1.5. Compilación

El proceso de compilación de Elixir comienza cuando se tiene uno o varios archivos con extensión *.ex*, los cuales son cargados en memoria y tomados como entrada por un lexer (o tokenizer). La salida de tokens producida será procesada por un parser que generará un AST, y este último se traducirá a bytecode, en particular en archivos de extensión *.beam*, los cuales son los que se ejecutarán en la máquina virtual.

La metaprogramación que Elixir nos ofrece, brinda una gran flexibilidad para hacer y deshacer con el lenguaje lo que se nos ocurra ya que justamente, mediante macros se puede modificar el AST para que el mismo sea "aceptado" a la hora de ser compilado y ejecutado. Para clarificar con un ejemplo, si trabajamos en un proyecto Elixir en el que sea necesario tener una sentencia de

iteración *while* que el lenguaje no la provee, se podría realizar fácilmente "metaprogramando" mediante macros para luego en el código, utilizar la sintaxis como si fuera una palabra reservada propia del lenguaje. (McCord, 2015)

Mix (Mix, 2020) forma una parte fundamental del mundo Elixir. Es una herramienta que se incluye con la instalación del lenguaje y proporciona tareas para crear, compilar, probar la aplicación, y administrar sus dependencias (en particular se lleva muy bien con *hex* (Hex, 2020), el manejador de paquetes del ecosistema). Cada proyecto Elixir tiene un archivo *mix.exs* en la raíz el cual especifica su configuración, como pueden ser las dependencias y la forma en que se compila, entre muchas otras posibles. En particular, la tarea de compilación de mix (*mix compile*) compila los archivos y genera un manifiesto de aplicación. Otra tarea interesante es la de formateo de código (*mix format*) la cual corrige los archivos con las reglas especificadas en un archivo *formatter.exs*, también autogenerado si creamos un proyecto utilizando la herramienta (*mix new*). Estas tareas son las que hacen que Mix sea tan dinámico y extensible ya que se puede extender su comportamiento agregando tareas propias, y dejando que estén disponibles para todos los que utilicen el proyecto, ejecutando *mix nombre_tarea*.

En términos de compilación Elixir, que sea un lenguaje dinámico significa que es casi nulo el chequeo de tipos cuando se ejecuta *mix compile*. Tener en cuenta que esta tarea está implícita en otras, por ejemplo también se ejecuta cuando se levanta la aplicación localmente, o cuando se realiza un *build* de la aplicación.

En la siguiente sección se introducirá a los tipos en los lenguajes de programación, así como los diferentes enfoques que se pueden encontrar a la hora de tiparlos, y así brindar una primera aproximación de por qué se consideran importantes los tipos en los lenguajes de programación.

2.2. Tipos en los lenguajes de programación

Remy (Rémy, 2013) menciona que los tipos juegan un papel central en el diseño de lenguajes de programación modernos. Son una descripción concisa y formal de como se comporta un fragmento de programa. Por ejemplo, *int* describe una expresión que se tipa como un entero, mientras que $int \rightarrow bool$ describe una función que toma un argumento entero y retorna un resultado booleano. Los programas deben comportarse según lo prescrito por sus tipos.

Los lenguajes de programación pueden ser divididos en tipados y no tipados. Cardelli (Cardelli, 2004) explica que los lenguajes en los que se le puede dar un tipo a las variables son lenguajes tipados. En estos lenguajes las variables de un programa pueden adoptar un rango de valores durante su ejecución, y el límite superior de dicho rango se denomina el tipo de la variable. Por otro lado, dice que los lenguajes que no restringen el rango de valores de las variables se denominan lenguajes no tipados. Se dice que éstos lenguajes no tienen tipos, o lo que es equivalente, tienen un único tipo que contiene todos los valores. En los lenguajes no tipados, las operaciones pueden aplicarse a argumentos inapropiados, dando

como resultado un error de ejecución o un comportamiento no deseado.

Un sistema de tipos es un componente de los lenguajes tipados que realiza un seguimiento de los tipos de todas las expresiones en un programa.

2.2.1. Sistema de tipos

La presente sección está basada en (Cardelli, 2004).

Los sistemas de tipos son generalmente formulados como colecciones de reglas para chequear la consistencia de los programas y evitar la ocurrencia de errores mientras éstos se ejecutan. El caso más claro de un error de ejecución es la aparición de una falla de forma inmediata, sin embargo existen errores más sutiles que no tienen síntomas inmediatos y generan problemas a largo plazo, como son los errores en la lógica o casteos inválidos.

Las nociones de sistemas de tipos son aplicables a todos los paradigmas de programación y especifica las reglas de tipos de un lenguaje de programación independientemente del algoritmo que se utilice luego para el chequeo de tipos. Es conveniente y útil desacoplar el sistema de tipos de los algoritmos de chequeo de tipos: los sistemas de tipos pertenecen a las definiciones de los lenguajes, mientras que los algoritmos pertenecen a los compiladores. Es más fácil explicar los aspectos de un lenguaje a través de su sistema de tipos que por el algoritmo usado por el compilador. Además, diferentes compiladores pueden usar diferentes algoritmos de chequeo para el mismo sistema de tipos.

El primer paso para formalizar un sistema de tipos para un lenguaje es describir su sintaxis. Para la mayoría de los lenguajes esto se reduce a describir la sintaxis de los tipos y términos. Los tipos expresan el conocimiento estático sobre los programas, mientras que los términos (sentencias, expresiones y otros fragmentos de programas) expresan el comportamiento algorítmico.

Otro aspecto fundamental a definir son los ambientes de variables, los cuales son usados para registrar los tipos de las mismas mientras se procesa un fragmento de programa. Este ambiente se corresponde con la tabla de símbolos de un compilador durante la fase de chequeo de tipos.

La descripción de los sistemas de tipos empiezan con la descripción de una colección de enunciados llamados juicios. Un juicio típico tiene la forma:

$$\Gamma \vdash \Im \qquad (\Im \text{ es una afirmación; las variables de } \Im \text{ están en } \Gamma)$$

Decimos que $\Im$ se deriva de Γ. Acá Γ es un ambiente de variables. La forma de las afirmaciones $\Im$ varía de juicio en juicio, pero todas las variables de $\Im$ deben estar declaradas en Γ.

Uno de los juicios más importantes, para el propósito del presente trabajo, es el que afirma que el término M tiene tipo A con respecto al ambiente de variables Γ, y tiene la forma:

$$\Gamma \vdash M : A \qquad (M \text{ tiene tipo } A \text{ en } \Gamma)$$

El siguiente paso es definir las reglas de tipado del lenguaje. Estas describen

la relación de que un término M tiene tipo A. Algunos lenguajes además requieren la relación de subtipo de la forma $A <: B$, que se detallará más adelante.

Las reglas afirman la validez de ciertos juicios basándose en otros juicios que ya son conocidos como válidos. Cada regla está formada por un conjunto de premisas (posiblemente vacío) y una conclusión, separados mediante una línea horizontal. La conclusión es válida cuando todas las premisas lo son.

$$\frac{\begin{array}{c}\Gamma_1 \vdash \Im_1 \\ \cdots \\ \Gamma_n \vdash \Im_n\end{array}}{\Gamma \vdash \Im} \text{ (prog tipado correcto)}$$

Una derivación se obtiene por la aplicación sucesiva de reglas de tipado. Las derivaciones se suelen presentar en forma arborescente. Un requerimiento fundamental en un sistema de tipos es que debe ser posible chequear que una derivación está correctamente construida. Un juicio válido es aquel que puede ser obtenido como raíz de una derivación para un sistema de tipos dado.

2.2.2. Tipado estático y dinámico

Para entender por qué el sistema de tipos es tan importante, así como para continuar entendiendo la motivación de este proyecto, lo primero que se debe hacer es considerar las ventajas y desventajas del tipado estático.

Uno de los beneficios más importantes del tipado estático es que la mayoría de los errores se encuentran en tiempo de compilación, en lugar de hacerlo en tiempo de ejecución como pasa con tipado dinámico. Con un lenguaje dinámico, se entrega código que parece funcionar perfectamente, pero luego un usuario puede caer en un caso borde donde el programa se rompe.

Con tipado estático, el código ni siquiera compila si existe un error de tipos. Esto también es un indicador de que los errores se encuentran donde ocurren primero, es decir, un objeto con el tipo incorrecto generará un problema de compilación en su origen, y no donde realmente se usa. Como resultado de esto, el tipado estático necesita muchas menos pruebas unitarias en torno al acuerdo de tipos entre métodos. Por supuesto que no excusa la falta total de pruebas, pero ciertamente reduce la carga.

En (Meijer y Drayton, 2004) se destaca una mejor documentación con tipado estático, por ejemplo, cuando se define la cantidad y tipo de parámetros de funciones. Por otro lado, el compilador tiene más oportunidad de optimización, por ejemplo, reemplazando llamadas virtuales por llamadas directas cuando el tipo exacto del receptor se conoce estáticamente. Destaca también mayor eficiencia en tiempo de ejecución y una mejor experiencia de desarrollo.

Y entonces, ¿tiene alguna desventaja el tipado estático? En primer lugar, el desarrollo es probable que sea más lento. Es más rápido llegar a una solución sobre determinado problema en un lenguaje con tipado dinámico. Otra desventaja, es que en lenguajes estáticos puede haber más código repetitivo, por ejemplo, en lenguajes como *C++* o *Java* se declara el tipo de casi todas las variables, aunque no es necesario en todos los lenguajes estáticos, como por ejemplo *Python*. Por

otro lado, los lenguajes dinámicos son ideales para la creación de prototipos de sistemas con requisitos cambiantes o desconocidos, o que interactúan con otros sistemas que cambian de manera impredecible.

2.2.3. Lenguajes seguros

Cardelli (Cardelli, 2004) expresa que un fragmento de programa es seguro si no causa errores de largo plazo, y un lenguaje seguro es aquel donde todos los fragmentos de programa son seguros.

Los lenguajes no tipados pueden reforzar su seguridad realizando verificaciones en tiempo de ejecución, mientras que los tipados pueden cumplir con la seguridad rechazando a aquellos programas que son potencialmente inseguros antes de ser ejecutados. Esto no quita que los lenguajes tipados también puedan realizar chequeos en tiempo de ejecución para reforzar su seguridad.

Se designa un conjunto de posibles errores de ejecución como errores prohibidos, y se dice que un programa tiene un buen comportamiento si no hace que ocurra ninguno de estos errores. Un lenguaje es fuertemente tipado si todos los programas (legales) tienen buen comportamiento.

Según Cardelli, un programa con un buen comportamiento es seguro. El objetivo de los sistemas de tipos generalmente es asegurar que todos los programas se comporten bien, distinguiendo entre programas bien tipados y mal tipados. Pero la realidad es que algunos lenguajes chequeados estáticamente no aseguran la seguridad completa. Estos lenguajes se consideran chequeados débilmente, lo que significa que estáticamente solo algunas operaciones inseguras son detectadas.

Algunos lenguajes como *C* son deliberadamente inseguros por consideraciones de *performance*: los chequeos necesarios para lograr la seguridad a veces se consideran demasiado caros. Además, la seguridad produce un comportamiento de detección de fallas en caso de errores de ejecución, lo que reduce el tiempo de depuración.

Por lo tanto, la elección entre un lenguaje seguro e inseguro puede estar relacionado con una compensación entre el tiempo de desarrollo y mantenimiento, y el tiempo de ejecución y la performance.

2.2.4. Tipado explícito e implícito

Anotar programas con tipos explícitos puede generar mucha redundancia. Los tipos incluso pueden volverse extremadamente engorrosos e incomprensibles cuando tienen que proporcionarse explícita y repetidamente. En algunos casos patológicos, incluso pueden aumentar el tamaño de los términos de forma no lineal. Esto crea la necesidad de un cierto grado de reconstrucción de tipos (también llamado inferencia de tipos), donde el programa fuente puede contener cierta información, pero no toda. (Rémy, 2013)

2.2.5. Inferencia de tipos

En un determinado sistema de tipos, un termino M es bien tipado para un ambiente Γ, si existe un tipo A tal que $\Gamma \vdash M : A$ es un juicio válido. Es decir, si al término M se le puede dar algún tipo.

El descubrimiento de la derivación de un término (y por lo tanto de su tipo) se llama problema de inferencia de tipos. Este problema depende mucho del sistema de tipos en cuestión, por lo que un algoritmo para la inferencia de tipos puede ser muy fácil, muy difícil o imposible de encontrar. Si se encuentra, el mejor algoritmo puede ser muy eficiente, o muy lento. Aunque los sistemas de tipos son expresados y a veces diseñados con un nivel alto de abstracción, su uso práctico depende de la disponibilidad de un buen algoritmo de inferencia de tipos que sea capaz de representar lo que se quiere.

2.2.6. Tipado gradual

Como se mencionó anteriormente, el tipado estático y el dinámico tienen ventajas y desventajas, y a la hora del desarrollo de un lenguaje de programación son tenidas en cuenta según cuál es el más adecuado para los problemas que intenta solucionar o que se quiere adaptar el lenguaje. De todas maneras, existen muchos intentos de combinar ambos beneficios en un mismo lenguaje, como es el ejemplo de TypeScript, entre otros.

En (Siek y Taha, 2006) se utiliza el término *gradual typing* (tipado gradual) para lenguajes funcionales, para definir a los sistemas de tipos que proveen tipado estático y dinámico en un mismo programa, y que el programador pueda controlar el grado de chequeo estático anotando con tipos los parámetros de las funciones. En (Siek y Taha, 2007) se presenta el tipado gradual para lenguajes orientados a objetos.

Además, se han presentado varios trabajos respecto a la unión del tipado estático y el dinámico que quedan por fuera del tipado gradual, como lo son soft typing (Cartwright y Fagan, 2004), success typing (Lindahl y Sagonas, 2006) o quasi-static typing (Thatte, 1989). En soft typing, se plantea que el verificador de tipos no necesita rechazar programas que contengan expresiones mal tipadas, sino que sea capaz de insertar chequeos explícitos en tiempo de ejecución, transformando programas mal tipados en programas de tipo correcto. Por otro lado en success typing, se incorpora la noción de subtipos y de nunca rechazar el uso de una función que no resulte en un error de tipos en tiempo de ejecución. Permite la inferencia de tipos sin ninguna declaración e incluso ante la ausencia de ciertos componentes del programa. De este abordaje, se desprende Dialyzer (Sagonas y Luna, 2008).

Siek (Siek y Taha, 2006) presenta un sistema de tipos formal, que soporta tipado gradual para lenguajes funcionales, proveyendo la flexibilidad del tipado dinámico cuando no se realizan anotaciones de tipos y los beneficios del chequeo estático cuando sí se realizan. Introduce el tipado gradual para una extensión del lambda cálculo con tipado simple con un tipo estáticamente desconocido, llamado ?. El chequeador de tipos se asegura de que las partes del programa

que son definidas con los tipos estáticos (que no contienen el tipo ?) gozan de la seguridad que el tipado estático garantiza, mientras que las partes anotadas con el tipo ? tienen la posibilidad de fallar en algún chequeo de tipos dinámico.

El tipado estático se basa en la propiedad de correctitud (o corrección) que refiere a que si el analizador dice que el programa está bien tipado, entonces efectivamente lo está y no va a dar un error de tipos durante la ejecución. En cambio, el tipado gradual se basa en que si el programa falla en tiempo de ejecución, entonces el problema radica en la parte dinámicamente tipada.

Por otro lado, se presenta la relación de consistencia de tipos ($\sim$). La misma es reflexiva y simétrica pero no transitiva, y sus principales reglas son:

$$? \sim t \qquad\qquad t \sim ? \qquad\qquad t \sim t$$

Para conocer la motivación detrás de esta relación, veamos el siguiente ejemplo. La siguiente expresión que toma un valor numérico y retorna su sucesor es rechazada cuando no se aplica a un valor numérico:

$$((\lambda\ (x : \textbf{number})\ (succ\ x))\ true) \tag{2.1}$$

Mientras que la siguiente, que toma un valor desconocido y retorna su sucesor será aceptada:

$$((\lambda\ (x : \textbf{?})\ (succ\ x))\ true) \tag{2.2}$$

No anotar el tipo es sinónimo de anotarlo con el tipo desconocido, el chequeador estático no lo tendrá en cuenta, por lo que la expresión (2.2) equivale a la siguiente:

$$((\lambda\ (x)\ (succ\ x))\ true) \tag{2.3}$$

La primera ecuación falla en tiempo de compilación porque el tipo **boolean** de *true* no es compatible con el tipo **number** de x. Las dos últimas ecuaciones no son chequeadas estáticamente, fallarán en tiempo de ejecución, cuando se realice el chequeo dinámico de tipos, al intentar aplicar la función *succ* al valor *true*.

En (Castagna y Lanvin, 2017) se propone un sistema de tipos para lenguajes funcionales con tipado gradual junto con la teoría de conjuntos para tipos. En particular, busca definir tipado gradual para el enfoque de subtipado semántico, donde los tipos son interpretados como conjuntos de valores. En (Castagna y cols., 2019) se extienden los beneficios presentados en el artículo anterior para una configuración polimórfica. En este articulo utilizan una nueva interpretación de los tipos graduales, donde el tipo desconocido ? funciona como una variable de tipo, y en donde cada ocurrencia de este tipo se puede considerar como un espacio para una variable de tipo. Definen una operación de *discriminación* donde cada ocurrencia del tipo ? se sustituye por una variable de tipo. Mediante la aplicación de la discriminación se mapea un tipo gradual polimórfico en un conjunto de tipos estáticos polimórficos. En particular, usan la discriminación para definir dos relaciones: la de subtipado y la de materialización. La relación de subtipado $t <: t'$ implica que una expresión de tipo t puede ser usada de forma segura en un lugar donde se espera una del tipo t'. La relación de materialización $t \preceq t'$ se da si y solo si t' es más preciso que t, es decir, t' es obtenida de reemplazar alguna ocurrencia de ? en t.

Algunos lenguajes han definido un sistema de tipos con tipado gradual, por ejemplo *Clojure*, *Python*, *PHP*, *C#* o *Erlang*. Algunos de estos son lenguajes con tipado dinámico en los que se busca obtener los beneficios del tipado estático, y otros al revés.

Como se mencionó anteriormente, *Erlang* cuenta con una herramienta de chequeo de tipos gradual (Gradualizer, 2020), donde se utiliza un enfoque similar al presentado en (Castagna y cols., 2019), utilizando al tipo *any()* como el tipo dinámico. Utiliza relaciones sobre los tipos como compatibilidad y subtipado, similares a las presentadas por este autor. En este caso, se busca que su uso no sea invasivo para el programador, por lo que en caso de no agregarse especificaciones de tipos no se hace ningún chequeo y el lenguaje permanece totalmente dinámico. A medida que se agregan especificaciones, mayor información se le brinda al sistema de tipos. Este enfoque no utiliza inferencia de tipos, por lo que los tipos están totalmente basados en las notaciones que el programador realiza. El sistema de tipos que se propone en este trabajo sigue el mismo enfoque, con la diferencia de que usamos el tipo `any` como *top type* (el súper tipo de la relación de subtipo) y como el tipo desconocido del *gradual typing*. Por otro lado, este sistema de tipos se diferencia de *Dialyzer* en que está basado en un sistema de tipos, por lo tanto, no es un analizador de tipos que intenta predecir errores en tiempo de ejecución.

Capítulo 3

Elixir con tipado estático

El presente capítulo describe el estudio realizado sobre el lenguaje Elixir desde una perspectiva centrada en sus tipos y la forma en que se chequean los mismos.

Elixir es un lenguaje con tipado dinámico, por lo que, en la búsqueda de contribuir al lenguaje en este sentido, la investigación se centra en cómo Elixir puede obtener los beneficios del tipado estático, sin perder su esencia y flexibilidad. Nuestro enfoque está inspirado en *gradual typing*.

3.1. Principales características

Se propone un sistema de tipos que permite realizar una verificación de tipos estática en un fragmento significativo de Elixir. La sintaxis es reducida a un conjunto básico del lenguaje para cubrir las expectativas del proyecto de grado.

El primer punto importante a destacar es que el sistema de tipos asume que el código Elixir a evaluar es válido, es decir, ha pasado los chequeos que realiza el compilador del lenguaje.

Es una solución no intrusiva, que busca mantener la esencia del lenguaje. Por lo tanto, no requiere de ninguna modificación en la sintaxis de Elixir y tampoco se producen cambios en el código analizado. La información de tipos se proporciona por medio de firmas de funciones que se declaran en términos de directivas específicas de Elixir. Es el programador quien decide qué partes del código deben ser chequeadas estáticamente.

Se introduce un sistema de tipos gradual, que hace un compromiso entre la seguridad del tipado estático y la flexibilidad del tipado dinámico, y se basa en las relaciones de precisión y subtipado. Para esto, se utiliza un tipo que contiene a los demás tipos: `any`, que a su vez será el tipo desconocido del *gradual typing*.

El sistema es compatible con código heredado, ya que permite la presencia de fragmentos de código no tipados. Las partes del código que no pueden ser chequeadas estáticamente por falta de información de tipos se verifican en

tiempo de ejecución tal como lo hace Elixir.

3.2. Sintaxis

Como se mencionó, la sintaxis de Elixir es más extensa que la detallada a continuación. Para representar este fragmento se utiliza EBNF, comenzando por los tipos y luego por los términos. Estos últimos son definidos desde los más simples hacia los más complejos (*bottom-up*) para que las definiciones formales y los ejemplos utilicen la sintaxis ya definida.

3.2.1. Tipos

Se definen los tipos que Elixir provee. Los básicos `integer`, `float`, `boolean`, `string` y `atom`, los tipos compuestos lista (`[`t`]`), tupla (`{`t_1`,...,`t_n`}`) y mapa (`%{`$t_{11} \Rightarrow t_{12}$`,...,`$t_{n1} \Rightarrow t_{n2}$`}`), y el tipo función (`(`t_1`,...,`t_n`)` $\rightarrow t$).

En las listas todos los elementos son de un mismo tipo, en las tuplas cada elemento puede tener su propio tipo, y en los mapas las claves son de un mismo tipo mientras que cada valor puede tener su propio tipo independiente. Por último, para el caso de las funciones cada parámetro tiene un tipo, y se tiene un único tipo de retorno.

En este trabajo se propone agregar el tipo `none` como el *bottom type*, o el tipo más específico, el cual es el subtipo de todos los tipos y el tipo `any` como el *top type*, o el tipo más genérico, del cual todos los tipos forman parte (súper tipo).

$$
\begin{array}{rcl}
t & ::= & \texttt{none} \\
 & | & \texttt{integer} \\
 & | & \texttt{float} \\
 & | & \texttt{boolean} \\
 & | & \texttt{string} \\
 & | & \texttt{atom} \\
 & | & \texttt{[}t\texttt{]} \\
 & | & \texttt{\{}t_1\texttt{,...,}t_n\texttt{\}} \\
 & | & \texttt{\%\{}t_{11} \Rightarrow t_{12}\texttt{,...,}t_{n1} \Rightarrow t_{n2}\texttt{\}} \\
 & | & \texttt{(}t_1\texttt{,...,}t_n\texttt{)} \rightarrow t \\
 & | & \texttt{any}
\end{array}
$$

3.2.2. Términos

3.2.2.1. Literales

Los literales se definen de forma similar a la mayoría de los lenguajes.

Se tienen números enteros (i), números flotantes (f), cadenas de caracteres (s), booleanos (b) y átomos (a). Los átomos o *atoms*, son constantes cuyo valor es su propio nombre y se anotan con dos puntos (:) seguidos de una letra y luego cualquier cantidad de caracteres. Existen constantes propias del lenguaje

que también son átomos, por ejemplo *nil*, utilizado en muchos otros lenguajes como el objeto vacío. Otras constantes que son tratadas como átomos en Elixir son *true* y *false* (aparte de ser parte de los booleanos). Para el presente trabajo estos dos últimos serán tratados únicamente como booleanos.

Algunos ejemplos de literales pueden ser:

```
:example    # atomo
123.543     # flotante
123         # entero
"example"   # cadena de caracteres
true        # booleano
```

Formalmente, se definen de la siguiente manera:

$$l \quad ::= \quad a \quad | \quad f \quad | \quad i \quad | \quad s \quad | \quad b$$

$$a \quad ::= \quad :id$$

$$s \quad ::= \quad \text{“}chr^*\text{”}$$

Donde $f \in \mathbb{R}$, $i \in \mathbb{Z}$ e $id \in Id$.

3.2.2.2. Variables

Las variables en Elixir se identifican con su nombre que debe comenzar con una letra minúscula y luego cualquier cantidad de caracteres. La convención del lenguaje es utilizar *snake case* (colocar "_" para separar elementos en un nombre).

Ejemplos:

```
x = :a
una_variable = 123.543
```

Se define:

$$x \quad ::= \quad id$$

3.2.2.3. Funciones

Las funciones en Elixir, al igual que las variables, se identifican con un nombre que comienza con una letra minúscula seguido de cualquier cantidad de caracteres. Se utiliza la misma convención que para las variables.

Se definen con la palabra reservada *def* (se pueden tener funciones privadas a un módulo, las cuales se definen como *defp*), seguidas del nombre, los parámetros entre paréntesis, y el cuerpo de la función dentro de un bloque que comienza y termina con las palabras reservadas *do* y *end* respectivamente.

Para invocar una función se hace con su nombre seguido de los parámetros entre paréntesis. Cabe destacar que en caso de que la función no pertenezca al mismo módulo, se dará su nombre completo (*fully qualified name*), para lo cual

se utiliza un prefijo que representa la ubicación de la definición de la función en la jerarquía de módulos.

```
defmodule A do
    defmodule B do
        def add1(x) do
            x + 1
        end
    end
end

defmodule C do
    def add1_duplicate(y) do
        A.B.add1(y)*2
    end
end
```

Para especificar el tipo de una función se utilizará la directiva *@spec* (que Elixir provee pero de la cual no hace uso, ya que es tratada como un comentario por el compilador), seguida del nombre de la función, el tipo de los parámetros y el tipo de retorno. Una función puede tener cualquier cantidad de parámetros pero un solo tipo de retorno.

El siguiente ejemplo muestra la especificación de una función que toma como parámetro un número entero, y retorna un número flotante

```
@spec dup_42(integer) :: float
def dup_42(x) do
    x * 42.0
end
```

Ejemplos de invocación a la función anterior:

```
dup_42(1) # 42.0
dup_42(2) # 84.0
```

Para el caso de una función que toma como parámetros un número entero (n) y una cadena de caracteres, y que retorna una lista que tiene como largo el entero y todos los elementos son la cadena de caracteres, se puede definir de la siguiente forma:

```
@spec replicate(integer, string) :: [string]

def replicate(0, _) do
    []
end

def replicate(n, value) do
    [value | replicate(n-1, value)]
end
```

Cuando se utiliza el guión bajo para un parámetro se está indicando que éste no va a ser utilizado.

El pattern-matching se realiza según el orden de definición de las funciones. Para este caso, la segunda definición de la función es aplicada cuando la anterior no es verdadera, es decir, cuando no se cumple que el entero es cero.

Otro aspecto interesante para mencionar es que las listas pueden ser construidas de la forma [*head* | *tail*], donde *head* representa al primer elemento o la cabeza, y *tail* al resto de la lista o la cola. En los patrones se verá como se pueden destruir (o desestructurar).

Ejemplos de invocación a la función anterior:

```
replicate(1, "ab") # ["ab"]
replicate(5, "ab") # ["ab", "ab", "ab", "ab", "ab"]
replicate(0, "ab") # []
```

Para especificar y definir una función que toma un mapa de tres elementos con cadenas de caracteres como claves y números enteros como valores, y retorna un boolean, se debe hacer de la siguiente forma:

```
@spec three_is_3(%{string => integer, string => integer,
    string => integer}) :: boolean
def three_is_3(map) do
    map["tres"] === 3
end
```

Ejemplos de invocación a la función anterior:

```
three_is_3(%{"uno" => 1, "dos" => 2, "tres" => 3}) # true
three_is_3(%{"uno" => 1, "dos" => 2, "cuatro" => 4}) # false
```

Formalmente queda definido como:

$$f \quad ::= \quad f; f \quad | \quad f\_spec \quad | \quad f\_def$$

Donde:

$$\begin{aligned}
f_name &::= id \\
f_spec &::= \texttt{@spec}\ f_name(t, ..., t)\ \texttt{::}\ t \\
f_def &::= \texttt{def[p]}\ f_name(p, ..., p)\ \texttt{do}\ e\ \texttt{end} \\
f_call &::= [(mod_name.)^*]\ f_name(e, ..., e)
\end{aligned}$$

Los p que se presentan son patrones, mientras que las e son expresiones. Ambos se presentan a continuación.

Las especificaciones de funciones no tienen por qué estar seguidas de su definición, pero sí deben estar antes (lo cual es chequeado por Elixir). Además como se vio, puede haber más de una definición para una misma especificación:

```
@spec dup(integer) :: integer
@spec dup_list_elem([integer]) :: [integer]

def dup(n) do
    n * 2
end

def dup_list_elem([]) do
    []
end

def dup_list_elem([head | tail]) do
    [dup(head) | dup_list_elem(tail)]
end
```

Ejemplos de usos de las funciones anteriores:

```
dup(2) # 4
dup_list_elem([1,2]) # [2,4]
dup_list_elem([]) # []
```

Como se mencionó anteriormente las partes del código que no pueden ser chequeadas estáticamente por falta de información de tipos se verifican en tiempo de ejecución tal como lo hace Elixir. Veamos el siguiente ejemplo:

```
@spec id(float) :: float
def id(n) do
    n
end

@spec pred(integer) :: integer
def pred(n) do
    n - 1
end

def succ(n) do
    n + 1
end
```

Las siguientes invocaciones tienen los siguientes resultados:

```
id(2)       # 2
id(2.0)     # 2.0
id("2")     # type error

pred(2)     # 1
pred(2.0)   # type error
pred("2")   # type error

succ(2)     # 3
succ(2.0)     # 3.0
succ("2")   # runtime error
```

Como se puede apreciar para las primeras dos funciones, al tenerse la especificación de sus tipos, se puede detectar el error de tipos en tiempo de compilación

cuando se llama con un valor con tipo incorrecto. Sin embargo, para la tercera función, como no se tiene la especificación de tipos para la función, el error se tiene en tiempo de ejecución. Por otro lado, para el caso de la segunda función se puede ver como una función que se define con tipo entero no puede ser llamada con un valor flotante, mientras que una función que se define con tipo flotante sí puede ser llamada con un entero. Esto lo veremos más adelante y se obtiene con el uso de la relación subtipo.

3.2.2.4. Patrones

Los patrones son términos utilizados en *parámetros*, *guardas* y *bindings*. Están compuestos por los literales (l), las variables (x), las listas (`[`p `|` p`]`), las tuplas (`{`p `, ..., `p`}`) y los mapas (`%{`mp `⇒` p `, ..., `mp `⇒` p`}`). También incluyen al *binding* (p `=` p), que como se verá tiene un comportamiento diferente que el de las expresiones, y el patrón *wildcard* (`_`) que como hemos visto en algunos ejemplos se utiliza para ignorar a un parámetro o argumento.

Los patrones se utilizan para desestructurar expresiones y realizar pattern matching. Entonces, si tenemos la tupla `{1, true}` como parámetro de una función, el cuerpo de la misma solo se ejecutará cuando se la invoque exactamente con ese patrón. Las guardas tienen un comportamiento similar. Un patrón de mapa coincide con un mapa si contiene las claves y sus respectivos valores también coinciden.

Formalmente, se definen:

$$
\begin{array}{lcl}
p & ::= & \texttt{\_} \\
 & | & l \\
 & | & x \\
 & | & p \texttt{ = } p \\
 & | & \texttt{[}p \texttt{ | } p\texttt{]} \\
 & | & \texttt{\{}p \texttt{ , ... , } p\texttt{\}} \\
 & | & \texttt{\%\{}mp \texttt{ ⇒ } p \texttt{ , ... , } mp \texttt{ ⇒ } p\texttt{\}}
\end{array}
$$

$$
\begin{array}{lcl}
mp & ::= & l \\
 & | & mp \texttt{ = } mp \\
 & | & \texttt{[}mp \texttt{ | } mp\texttt{]} \\
 & | & \texttt{\{}mp \texttt{ , ... , } mp\texttt{\}} \\
 & | & \texttt{\%\{}mp \texttt{ ⇒ } mp\texttt{, ... , } mp \texttt{ ⇒ } mp\texttt{\}}
\end{array}
$$

Hemos definido *mp*, que es una clase especial de patrones y es utilizada en las claves de los patrones mapas, las cuales no aceptan variables en la sintaxis definida por Elixir.

Ejemplos que lo ilustran:

```
a = 1
b = 2

%{a => 2} = %{1 => 2} # error
```

```
%{1 => a} = %{1 => 2} # correcto
%{1 => 2} = %{a => 2} # correcto
%{1 => 2} = %{1 => a} # correcto
```

En el primer ejemplo vemos como se tiene un error al utilizar una variable como clave de un patrón mapa.

3.2.2.5. Expresiones

Las expresiones están compuestas por todos los literales, variables, los operadores binarios y unarios, los condicionales, el case, el llamado a las funciones, el binding y la secuencia de expresiones, la cual tiene el tipo de su última expresión. El case y el binding son casos particulares dado que son expresiones compuestas también por patrones.

Las variables pueden ser ligadas a los valores a través del matcheo con bindings, y así utilizarlas más adelante en la secuencia:

```
x = 10 * 9
x + 10                                  # 100
```

Pueden ser ligadas muchas veces y su alcance es limitado al bloque donde fueron definidas. Por ejemplo, la siguiente secuencia de expresiones tipa a la tupla `{1,3}`:

```
x = 1
y = if true do x = 2; x + 1 else 4 end
{x,y}                                   # {1, 3}
```

La aplicación de mapas se define de la siguiente manera:

```
m = %{1 => false, 4 => "cuatro", 9 => true}

m[9]                                    # true
```

El `case` retorna el valor de la rama cuyo patrón coincida con el valor del selector, mientras que el `cond` tiene el mismo comportamiento del `if else` pero se pueden tener tantas ramas como se quiera, es decir, se plantean varias expresiones booleanas y se retorna el cuerpo de aquella que evalúa a verdadero procesando a las mismas secuencialmente.

Para simplificar, escribimos $\odot$ para agrupar a `if` y `unless` ya que su comportamiento es el mismo. Por otro lado, escribimos $\oplus$ para denotar a los operadores binarios: aritmeticos (`+, -, *, /`), booleanos (`and, or`), de comparación (`<, >, <=, >=, ==, !=, ===, !==`), concatenación de listas ($++$), sustracción de listas ($--$), y concatenación de cadena de caracteres (`<>`). Los operadores unarios $\ominus$ son la negación aritmética (`-`), y la negación booleana (`not`).

$$
\begin{array}{rcl}
e & ::= & l \\
& | & x \\
& | & \texttt{[}e\ \texttt{|}\ e\texttt{]} \\
& | & \texttt{\{}e\ \texttt{,} \ldots \texttt{,}\ e\texttt{\}} \\
& | & \texttt{\%\{}e \Rightarrow e\texttt{,} \ldots \texttt{,}\ e \Rightarrow e\texttt{\}} \\
& | & e\texttt{[}e\texttt{]} \\
& | & \odot\ e\ \texttt{do}\ e\ [\ \texttt{else}\ e\ ]\ \texttt{end} \\
& | & e\ \oplus\ e \\
& | & \ominus\ e \\
& | & \texttt{case}\ e\ \texttt{do}\ \{p \rightarrow e\}^{+}\ \texttt{end} \\
& | & \texttt{cond do}\ \{e \rightarrow e\}^{+}\ \texttt{end} \\
& | & f\_call \\
& | & p\ \texttt{=}\ e \\
& | & e\ \texttt{;}\ e
\end{array}
$$

Donde:

$$
\begin{array}{rcl}
\odot & \in & \{\texttt{if, unless}\} \\
\oplus & \in & \{\texttt{+, -, *, /, and, or, ++, -,} \\
& & \texttt{<>, <, >, <=, >=,} \\
& & \texttt{==, !=, ===, !==}\} \\
\ominus & \in & \{\texttt{not, -}\}
\end{array}
$$

En el Apéndice C se pueden encontrar ejemplos de programas correctos e incorrectos junto a su salida, según algunas de estas expresiones.

3.2.2.6. Programas y Módulos

Los módulos en Elixir se identifican con un nombre que debe comenzar con mayúscula y luego cualquier cantidad de caracteres, siguiendo la convención *camel case* (comenzar con mayúscula cada elemento en un nombre).

Un programa está compuesto por uno o varios módulos, o expresiones. No es posible tener un programa donde se tengan funciones y expresiones a un mismo nivel. Sí es correcto tener programas que únicamente contengan expresiones, lo que se conoce comúnmente como *script*. Ejemplos para ilustrar esto:

```
# a.ex
defmodule A do
    defmodule B do
        def plus_1(x) do
            x + 1
        end
    end

    defmodule C do
        @spec plus_2(integer):: integer
        def plus_2(x) do
            x + 2
        end
```

```
        def greater_3(x) do
            x > 3
        end
    end

    def plus_2(x) do
        x + 2
    end
end

# d.ex
defmodule D do
    def pass_exam(x, y) do
        z = A.B.plus_1(x) + A.plus_2(y)
        A.C.greater_3(z)
    end
end

# c.ex
IO.puts "pass exam script"
D.pass_exam(1, 5)
```

Los módulos pueden tener otros módulos anidados, un conjunto de funciones o expresiones.

El caso típico de un programa Elixir como lenguaje para desarrollo de aplicaciones web, es un conjunto de módulos compuestos por funciones.

Formalmente entonces, se define como:

$$
\begin{array}{lcl}
prog & ::= & m \mid e \\
m & ::= & m; m \mid \texttt{defmodule}\ mod\_name\ \texttt{do}\ \hat{m}\ \texttt{end} \\
mod\_name & ::= & Id \\
\hat{m} & ::= & m \mid e \mid f
\end{array}
$$

3.3. Sistema de Tipos

En esta sección se darán a conocer los juicios y las reglas que serán aplicadas para poder determinar si un programa tiene un tipado correcto.

Para determinar si un programa está bien tipado se realizan dos iteraciones sobre él. La primera, extrae un ambiente con todas las funciones para las cuales se especifica su tipo. La segunda, es un chequeo de tipos para todos los términos partiendo del ambiente de funciones generado en la iteración anterior. Un programa se define como correctamente tipado si, luego de realizar la primera iteración, en la segunda iteración todos los términos cumplen con las reglas de chequeo. Que se realicen dos iteraciones hace que sea necesario definir dos tipos de reglas, las reglas de recorrida y las reglas de chequeo.

Se utilizará la letra Δ para representar el ambiente de funciones, y la letra Γ para representar el ambiente de variables. Las funciones para las que se especifica

su tipo se agregarán a Δ, y las variables utilizadas se agregarán al ambiente Γ luego de que se conozca su tipo. A partir de esto, se chequeará que el posterior uso de funciones y variables sea con un tipo correcto según el tipo con el que fueron agregadas al respectivo ambiente. Ambos ambientes se inicializan como el conjunto vacío ($\varnothing$), es decir, no existe ninguna variable ni función predefinida.

Por otro lado, se utilizará un prefijo ρ que dirá en todo momento en que ubicación de la jerarquía de módulos se está. Este prefijo será utilizado para agregar funciones al ambiente, esto es, las funciones son agregadas al ambiente por su nombre completo (*fully qualified name*). Por ejemplo, si tenemos una función f dentro del módulo B, que a su vez se encuentra dentro de otro módulo A, tenemos que su prefijo será $A.B$, entonces la función f será agregada al ambiente como $A.B.f$ con sus correspondientes tipos de parámetros y retorno. Dicho prefijo se inicializa como vacío, ϵ.

3.3.1. Subtipado

El subtipado captura la noción intuitiva de inclusión entre tipos, donde los tipos son vistos como una colección de valores. Un elemento de un tipo puede ser considerado también como elemento de cualquiera de sus súper tipos. Esto permite que el elemento pueda ser usado flexiblemente en diferentes contextos.

Cuando consideramos una relación de subtipo, normalmente se agrega un nuevo juicio $\Gamma \vdash A <: B$ diciendo que A es subtipo de B. La intuición es que cualquier elemento de A es un elemento de B o, cualquier programa de tipo A es un programa de tipo B. De esta manera, se pueden tener por ejemplo, números enteros en contextos de números reales.

3.3.1.1. Relación subtipo

La relación de subtipo, denotada como $<:$, es usada para representar el subtipado en nuestro sistema de tipos. Esta relación como se explicó en el párrafo anterior, por ejemplo, permite que todas las funciones que aceptan `float` también aceptan `integer`.

La relación de subtipado es reflexiva y transitiva:

$$\frac{}{t <: t}\ (\text{ST\_REFL}) \qquad \frac{t_1 <: t_2 \quad t_2 <: t_3}{t_1 <: t_3}\ (\text{ST\_TRANS})$$

El tipo `none` es subtipo de todos los tipos, mientras que `any` es supertipo de todos. Por otro lado, `integer` es subtipo de `float`:

$$\frac{}{\texttt{none} <: t}\ (\text{ST\_NONE}) \qquad \frac{}{t <: \texttt{any}}\ (\text{ST\_ANY})$$

$$\frac{}{\texttt{integer} <: \texttt{float}}\ (\text{ST\_NUM})$$

Para los tipos de las listas y tuplas los constructores de tipos preservan la relación de subtipo:

$$\frac{t <: u}{[t] <: [u]}\ (\text{ST\_LIST})$$

$$\frac{t_1 <: u_1 \quad \cdots \quad t_n <: u_n}{\{t_1, \ldots, t_n\} <: \{u_1, \ldots, u_n\}}\ (\text{ST\_TUPLE})$$

La siguiente regla indica la relación subtipo entre mapas:

$$\frac{k' <: k \qquad u_1 <: u'_1 \quad \cdots \quad u_n <: u'_n}{\%\{k \Rightarrow u_1, \ldots, k \Rightarrow u_{n+m}\} <: \%\{k' \Rightarrow u'_1, \ldots, k' \Rightarrow u'_n\}}\ (\text{ST\_MAP})$$

Los mapas son covariantes en los tipos de los elementos (o valores) ($u <: u'$) y contravariantes en los tipos de las claves ($k' <: k$). Además, un mapa con más entradas clave-valor es subtipo de uno con menos (si los que sí se corresponden son subtipos).

La siguiente regla indica la relación entre funciones. Es similar a la de mapas, ya que las funciones son covariantes en el tipo de retorno y contravariantes en los tipos de los parámetros:

$$\frac{u_1 <: t_1 \quad \cdots \quad u_n <: t_n \qquad t <: u}{(t_1, \ldots, t_n) \to t <: (u_1, \ldots, u_n) \to u}\ (\text{ST\_FUN})$$

3.3.1.2. Relación subtipo limitado

La relación de subtipado limitado, denotada como $<|$, es una variante de la relación de subtipado definida anteriormente, con la diferencia de que se excluye a `any` como supertipo de los demás.

Esta relación tiene el objetivo de evitar la generalización de tipos de expresiones que no son correctamente tipadas al tipo `any` por ser el súper tipo. Esto se necesita porque en nuestro sistema de tipos `any` es tanto el tipo desconocido de *gradual typing* como el súper tipo. De esta manera, por ejemplo, se logra la construcción de listas heterogéneas, ya que sino una lista de diferentes tipos daría como súper tipo a `any` y no fallaría en tiempo de compilación. Más adelante se detallan más aplicaciones y ejemplos.

$$\frac{}{t <| t}\ (\text{LS\_REFL}) \qquad \frac{t_1 <| t_2 \quad t_2 <| t_3}{t_1 <| t_3}\ (\text{LS\_TRANS})$$

$$\frac{}{\texttt{none} <| t}\ (\text{LS\_NONE}) \qquad \frac{}{\texttt{integer} <| \texttt{float}}\ (\text{LS\_NUM})$$

$$\frac{t <| u}{[t] <| [u]}\ (\text{LS\_LIST})$$

$$\frac{t_1 <| u_1 \quad \cdots \quad t_n <| u_n}{\{t_1, \ldots, t_n\} <| \{u_1, \ldots, u_n\}}\ (\text{LS\_TUPLE})$$

$$\frac{k' <: k \qquad u_1 <| u'_1 \quad \cdots \quad u_n <| u'_n}{\%\{k \Rightarrow u_1, \ldots, k \Rightarrow u_{n+m}\} <| \%\{k' \Rightarrow u'_1, \ldots, k' \Rightarrow u'_n\}}\ (\text{LS\_MAP})$$

$$\frac{u_1 <| t_1 \quad \cdots \quad u_n <| t_n \qquad t <| u}{(t_1, \ldots, t_n) \to t <| (u_1, \ldots, u_n) \to u}\ (\text{LS\_FUN})$$

3.3.1.3. Relación de precisión

La relación de precisión dice que $u \lll t$ cuando u resulta de cambiar alguna ocurrencia de `any` en t por otro tipo.

Esta relación tiene el objetivo de poder utilizar expresiones que tienen tipo `any` en las distintas operaciones con el tipo que se requiere. Por ejemplo, al realizar una suma que tenga como operando una función para la cual no se conoce su tipo de retorno. Más adelante veremos en más detalle su uso y otros ejemplos.

La precisión es reflexiva (PR_REFL). Además, la regla (PR_ANY) indica que todos los tipos son más precisos que `any`. El resto de las reglas son de construcción. A diferencia del subtipado, la relación para mapas y funciones es covariante en todas sus posiciones.

Un tipo no es concretizado a cualquier tipo arbitrario, sino a uno que respeta su estructura. Por ejemplo, `{integer, boolean}` se relaciona mediante la relación de precisión con sí mismo, `any`, `{any, boolean}`, `{integer, any}` y `{any, any}`.

$$\frac{}{t \lll t}\ (\text{PR\_REFL}) \qquad \frac{}{t \lll \texttt{any}}\ (\text{PR\_ANY})$$

$$\frac{t \lll u}{[t] \lll [u]}\ (\text{PR\_LIST})$$

$$\frac{t_1 \lll u_1 \quad \cdots \quad t_n \lll u_n}{\{t_1, .., t_n\} \lll \{u_1, .., u_n\}}\ (\text{PR\_TUPLE})$$

$$\frac{k \lll k' \qquad u_1 \lll u'_1 \quad \cdots \quad u_n \lll u'_n}{\%\{k \Rightarrow u_1, \ldots, k \Rightarrow u_n\} \lll \%\{k' \Rightarrow u'_1, \ldots, k' \Rightarrow u'_n\}}\ (\text{PR\_MAP})$$

$$\frac{t_1 \lll u_1 \quad \cdots \quad t_n \lll u_n \qquad t \lll u}{(t_1, \ldots, t_n) \to t \lll (u_1, \ldots, u_n) \to u}\ (\text{PR\_FUN})$$

3.3.1.4. Subsumption y Downcast

Utilizaremos dos reglas para generalizar y especificar el tipo de un término, y así poder utilizarlo en diferentes contextos.

La regla *subsumption* se define para generalizar el tipo de un término a un supertipo de éste. Para esto se utiliza la relación de subtipo limitado que se presentó:

$$\frac{\Gamma \vdash term : t \qquad t <| u}{\Gamma \vdash term : u}\ (\text{SUB})$$

Se utiliza la relación de subtipado limitado para restringir la generalización a tipos con ocurrencias del tipo `any`, evitando que algunas expresiones mal tipadas

puedan ser chequeadas correctamente. Si permitiéramos una forma de subsumption no restringida, se aceptarían expresiones mal tipadas como la siguiente, donde ambas ramas del `if` pueden ser generalizadas al tipo `any`:

```
if c do 3 else 'a' # mal tipado
```

La regla relacionada a esta expresión es definida más adelante pero, para entender el ejemplo, basta con saber que se espera que tanto el primer bloque (*3*) como el segundo (*'a'*) tengan un mismo tipo. En este caso, si generalizamos los tipos de ambas expresiones sin limitarlos ($<:$) ambas lo harían al tipo `any` por ser el supertipo en común y la regla no fallaría. En cambio, utilizando el subtipado limitado ($<|$) esta regla falla porque los tipos `string` e `integer` no tienen un supertipo en común. El siguiente caso es válido, donde existe un supertipo con el que derivan ambas expresiones (`float`):

```
if c do 1 else 1.2 end
```

De esta manera, también podemos rechazar listas heterogéneas, mapas con claves de diferentes tipos, etc.

Por otro lado, la regla *downcast* representa la reducción a tipos más específicos mediante la aplicación de la relación de precisión que se introdujo anteriormente:

$$\frac{\Gamma \vdash term : t \qquad u \lll t}{\Gamma \vdash term : u} \text{ (DOWN)}$$

Una expresión tipa a un tipo (u) si lo hace para un tipo más genérico (t).

Como se mencionó al presentar la relación de precisión, el objetivo es poder utilizar expresiones de tipos que contienen al tipo `any` para las distintas operaciones. El tipo `any` es nuestro tipo dinámico y algunas expresiones pueden tener este tipo, por ejemplo, una función para la cual no se especifica su tipo o una función para la cuál se especifica que su tipo de retorno es `any`.

En el siguiente caso de uso se suma el valor de una función con un entero:

```
length([1]) + 3
```

Si no se especificó el tipo de la función *length*, entonces, para nuestro sistema de tipos su tipo de retorno será el tipo dinámico `any`. Sin embargo, las operaciones aritméticas sólo aceptan expresiones que su tipo sea un subtipo de `float`, y no queremos dar un error en tiempo de compilación al utilizar el tipo dinámico. Entonces, es necesario reducir el tipo `any` a un subtipo de `float` (en este caso `integer`) y así mantener la flexibilidad al no tipar funciones. Notar que esto podría desembocar en un error de ejecución si la función evalúa a un valor que tiene un tipo distinto al esperado por la operación. Más adelante veremos las reglas específicas para estos casos de uso.

3.3.2. Juicios

3.3.2.1. De programa correctamente tipado

Los siguientes juicios expresan que un programa compuesto por módulos m o expresiones e está correctamente tipado:

$$\vdash^{p} m \qquad \text{(j. prog mod correcto)}$$

$$\vdash^{p} e \qquad \text{(j. prog exp correcto)}$$

3.3.2.2. De recorrida

Los siguientes juicios indican que la recorrida para módulos m, funciones f o expresiones e, con un ambiente inicial de funciones Δ y un prefijo ρ, genera un ambiente de funciones Δ':

$$\Delta; \rho \vdash^{r} m \Rightarrow \Delta' \qquad \text{(j. recorrida mod)}$$

$$\Delta; \rho \vdash^{r} f \Rightarrow \Delta' \qquad \text{(j. recorrida func)}$$

$$\Delta; \rho \vdash^{r} e \Rightarrow \Delta' \qquad \text{(j. recorrida exp)}$$

3.3.2.3. De chequeo

Los siguientes juicios indican que el chequeo de módulos m o funciones f, con un ambiente inicial de funciones Δ y un prefijo ρ, tiene tipado correcto:

$$\Delta; \rho \vdash^{ch} m \qquad \text{(j. chequeo mod)}$$

$$\Delta; \rho \vdash^{ch} f \qquad \text{(j. chequeo func)}$$

3.3.2.4. De tipado

El siguiente juicio indica que el tipado de expresiones e con un ambiente inicial de funciones Δ, un ambiente inicial de variables Γ y un prefijo ρ, tiene tipo t y genera un ambiente de variables Γ':

$$\Delta; \Gamma; \rho \vdash^{t} e : t \Rightarrow \Gamma' \qquad \text{(j. tipado exp)}$$

El siguiente juicio indica que el tipado de patrones p con un ambiente inicial de variables Γ, tiene tipo t y genera un ambiente de variables Γ':

$$\Gamma \vdash^{tp} p : t \Rightarrow \Gamma' \qquad \text{(j. tipado pat)}$$

El siguiente juicio indica que el tipado de claves de patrones mapas kp con un ambiente inicial de variables Γ, tiene tipo t:

$$\Gamma \vdash^{ktp} kp : t \qquad \text{(j. tipado claves)}$$

3.3.3. Reglas programa correctamente tipado

Un programa compuesto de módulos tiene tipado correcto si luego de recorrerlo y obtener todas las funciones, se cumplen las reglas de chequeo para éste. Para representar lo dicho anteriormente se utiliza la siguiente regla:

$$\frac{\varnothing; \epsilon \vdash^{r} m \Rightarrow \Delta \quad \Delta; \epsilon \vdash^{ch} m}{\vdash^{p} m} \text{ (P\_M)}$$

Esta regla está definida por el juicio (j. prog mod correcto), la recorrida del programa está definida por el juicio (j. recorrida mod) y el chequeo de tipos del módulo se define por el juicio (j. chequeo mod).

Un programa compuesto de expresiones (*script*) tiene tipado correcto si se cumplen las reglas de chequeo:

$$\frac{\varnothing; \varnothing; \epsilon \vdash^{t} e : t \Rightarrow \Gamma}{\vdash^{p} e} \text{ (P\_E)}$$

En esta regla, definida por el juicio (j. prog exp correcto), no es necesario que se recorra el programa en busca de funciones porque no pueden ser declaradas en este tipo de programas. Por lo tanto, es suficiente que el programa sea correctamente tipado, definido por el juicio (j. tipado exp), para un ambiente de funciones inicial vacío y tenga un tipo genérico t.

3.3.4. Reglas de recorrida

3.3.4.1. Módulos

Los módulos son recorridos en busca de las funciones que son definidas dentro de ellos. Antes de definir cómo se obtienen las funciones dentro de un módulo debemos definir cómo son tipados los módulos en secuencia, para lo cual se utiliza la siguiente regla:

$$\frac{\Delta; \rho \vdash^{r} m_1 \Rightarrow \Delta^1 \quad \Delta^1; \rho \vdash^{r} m_2 \Rightarrow \Delta^2}{\Delta; \rho \vdash^{r} m_1; m_2 \Rightarrow \Delta^2} \text{ (R\_MS)}$$

La regla, definida por el juicio (j. recorrida mod), expresa la recorrida de secuencias de módulos a partir de un ambiente de funciones y un prefijo, extendiendo el ambiente de funciones a partir de recorrer ambas partes de la secuencia. Notar que ambos argumentos son recorridos con el mismo prefijo inicial, el cual es utilizado para definir la jerarquía de módulos.

La siguiente regla indica cómo se recorre un módulo en particular:

$$\frac{\Delta; \rho.m\_name \vdash^r m \Rightarrow \Delta^1}{\Delta; \rho \vdash^r \texttt{defmodule}\ m\_name\ \texttt{do}\ m\ \texttt{end} \Rightarrow \Delta^1}\ (\text{R\_M})$$

La recorrida de un módulo consta de recorrer su contenido. Esto se hace con el ambiente de funciones inicial Δ y con el prefijo formado por el inicial (ρ) y el nombre del módulo a recorrer, extendiendo al ambiente con el que concluye la regla, Δ'.

3.3.4.2. Funciones

Las funciones son los elementos que se agregan al ambiente. Todas las funciones especificadas serán agregadas, más allá de ser públicas o privadas. El control de que en un módulo no se incluya una función privada de otro módulo, lo realiza Elixir en su proceso de compilación, por lo que no es necesario tenerlo en cuenta en estas reglas.

En primer lugar, al igual que para los módulos, se define la regla para la recorrida de secuencia de funciones:

$$\frac{\begin{array}{c}\Delta; \rho \vdash^r f_1 \Rightarrow \Delta^1 \\ \Delta^1; \rho \vdash^r f_2 \Rightarrow \Delta^2\end{array}}{\Delta; \rho \vdash^r f_1; f_2 \Rightarrow \Delta^2}\ (\text{R\_FS})$$

La regla, definida por el juicio (j. recorrida func), es similar a la de módulos y expresa la recorrida de secuencias de funciones a partir de un ambiente de funciones y un prefijo, extendiendo el ambiente de funciones a partir de recorrer ambas partes de la secuencia.

Las siguientes dos reglas indican cómo es recorrida una función en particular, siendo las dos más significantes de este conjunto de reglas.

Por un lado, se tiene la especificación de la función. En esta regla se agrega una función al ambiente. Tener en cuenta que las funciones se identifican por su nombre y aridad. Esto es porque Elixir identifica a las funciones de esta forma (puede haber funciones con el mismo nombre si la cantidad de parámetros es distinta), por lo tanto una función será agregada al ambiente siempre y cuando no exista ya una función con esas características:

$$\frac{(\rho.f\_name, n) \notin \Delta}{\Delta; \rho \vdash^r \texttt{@spec}\ f\_name(t_1, ..., t_n)\ \texttt{::}\ t \Rightarrow \Delta[(\rho.f\_name, n) \mapsto \texttt{(}t_1\texttt{,} \ldots \texttt{,}t_n\texttt{)} \to t]}\ (\text{R\_FSPEC})$$

La regla expresa que una función es agregada al ambiente de funciones con el prefijo inicial y los tipos especificados.

La notación utilizada para decir que la función con prefijo ρ, nombre f_name, $t_1, ..., t_n$ como tipos de los parámetros y t como tipo de retorno es agregada al ambiente de funciones Δ es la siguiente:

$$\Delta[(\rho.f\_name, n) \mapsto \texttt{(}t_1\texttt{,} \ldots \texttt{,}t_n\texttt{)} \to t] \qquad (\text{N\_ADD\_FN})$$

Por otro lado, la siguiente notación es utilizada en la regla para expresar que una función con prefijo ρ, nombre *f_name* y una cantidad n de parámetros no se encuentra en el ambiente de funciones Δ:

$$(\rho.f\_name, n) \notin \Delta \qquad \text{(N\_N\_IN\_FN)}$$

Se define la siguiente regla que representa la recorrida para definiciones de funciones:

$$\frac{}{\Delta; \rho \vdash^r \texttt{def[p]}\ f\_name(p_1, ..., p_n)\ \texttt{do}\ e\ \texttt{end} \Rightarrow \Delta} \text{(R\_FDEF)}$$

La regla expresa que las definiciones de funciones no modifican el ambiente de funciones en la recorrida.

A continuación se muestra un ejemplo de código donde se tienen dos módulos en secuencia y dentro de cada uno se especifica y se define una función:

```
defmodule A do
    @spec plus_one(integer) :: integer
    def plus_one(x) do
        x + 1
    end
end

defmodule B do
    @spec plus_two(integer) :: float
    def plus_two(x) do
        x + 2.0
    end
end
```

Veamos algunas derivaciones para entender un poco más estas reglas. Las siguientes son derivaciones para especificaciones de funciones dentro de los módulos:

$$\frac{(A.plus\_one, 1) \notin \Delta}{\varnothing; A \vdash^r \texttt{@spec}\ plus\_one(\texttt{integer})\ \texttt{::}\ \texttt{integer} \Rightarrow \{(A.plus\_one, 1) \mapsto \texttt{(integer)} \rightarrow \texttt{integer}\}} \text{(R\_FSPEC)}$$

$$\frac{(B.plus\_two, 1) \notin \Delta}{\Delta = \{(A.plus\_one, 1) \mapsto \texttt{(integer)} \rightarrow \texttt{integer}\}; B \vdash^r \texttt{@spec}\ plus\_two(\texttt{integer})\ \texttt{::}\ \texttt{float} \Rightarrow \Delta[(B.plus\_two, 1) \mapsto \texttt{(integer)} \rightarrow \texttt{float}]} \text{(R\_FSPEC)}$$

El siguiente es un ejemplo de cómo se deriva un módulo en particular:

$$\frac{\varnothing; A \vdash^r \texttt{@spec}\ plus\_one(\texttt{integer})\ \texttt{::}\ \texttt{integer}\ ... \Rightarrow \Delta = \{(A.plus\_one, 1) \mapsto \texttt{(integer)} \rightarrow \texttt{integer}\}}{\varnothing; \epsilon \vdash^r \texttt{defmodule}\ A\ \texttt{do}\ \texttt{@spec}\ plus\_one(\texttt{integer})\ \texttt{::}\ \texttt{integer}\ ...\ \texttt{end} \Rightarrow \Delta} \text{(R\_M)}$$

Notar que no se agrega la definición de la función en la derivación porque no aporta en este ejemplo y así evitar que se vuelva difícil de entender (la representamos con puntos suspensivos).

Por último, la secuencia de módulos se deriva de la siguiente forma:

$$\frac{\begin{array}{l}\varnothing; \epsilon \vdash^r \texttt{defmodule}\ A\ \texttt{do}\ \ldots\ \texttt{end} \\ \qquad \Rightarrow \Delta^1 = \{(A.plus\_one, 1) \mapsto \texttt{(integer)} \to \texttt{integer}\} \\ \Delta^1; \epsilon \vdash^r \texttt{defmodule}\ B\ \texttt{do}\ \ldots\ \texttt{end} \\ \qquad \Rightarrow \Delta^2 = \Delta^1[(B.plus\_two, 1) \mapsto \texttt{(integer)} \to \texttt{float}]\end{array}}{\varnothing; \epsilon \vdash^r \texttt{defmodule}\ A\ \texttt{do}\ \ldots\ \texttt{end}; \texttt{defmodule}\ B\ \texttt{do}\ \ldots\ \texttt{end} \Rightarrow \Delta^2}\ \text{(R\_MS)}$$

3.3.4.3. Expresiones

Las expresiones no modifican el ambiente de funciones en la recorrida. La siguiente regla representa lo mencionado:

$$\frac{}{\Delta; \rho \vdash^r e \Rightarrow \Delta}\ \text{(R\_E)}$$

3.3.5. Reglas de chequeo

3.3.5.1. Módulos

La siguiente regla representa el chequeo de tipos de una secuencia de módulos:

$$\frac{\begin{array}{c}\Delta; \rho \vdash^{ch} m_1 \\ \Delta; \rho \vdash^{ch} m_2\end{array}}{\Delta; \rho \vdash^{ch} m_1; m_2}\ \text{(CH\_MS)}$$

La regla, definida por el juicio (j. chequeo mod), expresa que el chequeo de tipos para la secuencia de módulos tiene tipo correcto si las dos partes de la secuencia lo hacen para el mismo ambiente y prefijo.

La siguiente regla representa el chequeo de tipos de la definición de un módulo:

$$\frac{\Delta; \rho.m\_name \vdash^{ch} m}{\Delta; \rho \vdash^{ch} \texttt{defmodule}\ m\_name\ \texttt{do}\ m\ \texttt{end}}\ \text{(CH\_MM)}$$

Esta regla expresa que la definición de un módulo tiene tipo correcto si se chequea que el contenido del módulo tiene tipo correcto con el prefijo extendido por el prefijo inicial (ρ) y el nombre del módulo.

Esta regla también debe ser declarada de la siguiente forma, la lógica es la misma pero el contenido del módulo son funciones y se define con el juicio (j. chequeo func):

$$\frac{\Delta; \rho.m\_name \vdash^{ch} f}{\Delta; \rho \vdash^{ch} \texttt{defmodule}\ m\_name\ \texttt{do}\ f\ \texttt{end}}\ \text{(CH\_MF)}$$

También, debe ser declarada de la siguiente forma, la lógica es la misma pero el contenido del módulo son expresiones y se define con el juicio (j. tipado exp):

$$\frac{\Delta; \varnothing; \rho.m\_name \vdash^{t} e : t \Rightarrow \Gamma}{\Delta; \rho \vdash^{ch} \texttt{defmodule}\ m\_name\ \texttt{do}\ e\ \texttt{end}}\ \text{(CH\_ME)}$$

En este caso, como el contenido del módulo es una expresión (o una secuencia de éstas), la regla dice que existe un tipo t tal que la expresión tiene tipo t, y se genera un nuevo ambiente de variables (Γ) que no modifica la conclusión de la regla. Además, el chequeo de tipos para expresiones se realiza con un ambiente de variables vacío.

3.3.5.2. Funciones

La siguiente regla representa el chequeo de tipos de una secuencia de funciones:

$$\frac{\begin{array}{c}\Delta; \rho \vdash^{ch} f_1 \\ \Delta; \rho \vdash^{ch} f_2\end{array}}{\Delta; \rho \vdash^{ch} f_1; f_2}\ \text{(CH\_FS)}$$

Esta regla, definida por el juicio (j. chequeo func), es similar a la de módulos y expresa que la secuencia de funciones tiene tipo correcto si las dos partes de la secuencia lo hacen para el mismo ambiente de funciones y prefijo.

La siguiente regla representa el chequeo de tipos para la definición de funciones, para el caso en el que se haya provisto una especificación de tipo para la misma:

$$\frac{\begin{array}{c}(\rho.f\_name, n) \in \Delta \\ \Delta(\rho.f\_name, n) = (t_1, \ldots, t_n) \to t \\ \varnothing \vdash^{tp} p_1 : t'_1 \Rightarrow \Gamma^1 \\ \ldots \\ \Gamma^{n-1} \vdash^{tp} p_n : t'_n \Rightarrow \Gamma^n \\ \Delta; \Gamma^n; \rho \vdash^{t} e : t \Rightarrow \Gamma' \\ t'_1 <: t_1\ \ldots\ t'_n <: t_n\end{array}}{\Delta; \rho \vdash^{ch} \texttt{def[p]}\ f\_name(p_1, \ldots, p_n)\ \texttt{do}\ e\ \texttt{end}}\ \text{(CH\_FN)}$$

Esta regla dice que, para el chequeo de la definición de una función, en primer lugar se chequea si la función con prefijo ρ, nombre f_name y una cantidad n de parámetros se encuentra en el ambiente de funciones Δ, para lo cual utilizamos la siguiente notación:

$$(\rho.f\_name, n) \in \Delta \qquad \text{(N\_IN\_FUN)}$$

Además, se utiliza la siguiente para representar la obtención de los tipos de los parámetros y de retorno con los que la función se encuentra en el ambiente:

$$\Delta(\rho.f\_name, n) = (t_1, \ldots, t_n) \to t \qquad \text{(N\_TY\_FUN)}$$

Los parámetros con los que se define la función deben ser subtipos de los tipos con los que la función está definida en el ambiente, agregando las variables que definen al ambiente de variables. Como se puede observar, no se utiliza subsumption, esto está motivado por el siguiente ejmeplo:

```
@spec is_one(any) :: string
def is_one(1) do "uno" end
def is_one(_) do "otra cosa" end
```

Si se usara subsumption (la cual está definida con la relación de subtipo limitado) no se podría construir la primera definición porque el parámetro del tipo `integer` no podría ser generalizado al tipo `any` por la limitación de la relación. Se tomó la decisión de que este tipo de funciones puedan ser creadas, utilizando el subtipado sin limitaciones.

El cuerpo de la función debe tener como tipo de retorno el que tiene definida en el ambiente, utilizando el ambiente de variables generado a partir del tipado de los parámetros.

Es necesario notar que si una variable se declara en un parámetro de la función con un tipo no puede ser declarada en otro parámetro con otro tipo, es por eso que el Γ con las ligaduras de los patrones se va pasando entre los parámetros. El siguiente es un ejemplo de una definición que no es correcta:

```
@spec say_hi(integer, string) :: string
def say_hi(a, a) do      # mal tipado
    "Hi " <> a
end
```

Mientras que el siguiente sí es un ejemplo válido:

```
@spec equal?(integer, integer) :: string

def equal?(x, x) do
    "Son iguales"
end

def equal?(_, _) do
    "Son distintos"
end
```

Ejemplos de llamada a la función anterior:

```
equal?(1,1) # "Son iguales"

equal?(1,2) # "Son distintos"
```

Para el caso en que la función no se encuentre en el ambiente de funciones Δ, es decir, no tiene una especificación de tipos (para lo cual se utiliza N_N_IN_FN, se utiliza la siguiente regla que expresa que no se realiza un chequeo sobre la definición:

$$\frac{(\rho.f\_name, n) \notin \Delta}{\Delta; \rho \vdash^{ch} \texttt{def[p]}\ f\_name(p_1, ..., p_n)\ \texttt{do}\ e\ \texttt{end}}\ \text{(CH\_FN\_NT)}$$

3.3.6. Reglas de tipado

3.3.6.1. Patrones

Las reglas para patrones están definidas por el juicio (j. tipado pat).

Las reglas de tipado de los literales son representadas de la siguiente forma:

$$\frac{}{\Gamma \vdash^{tp} f : \texttt{float} \Rightarrow \Gamma}\ \text{(TP\_F)} \qquad \frac{}{\Gamma \vdash^{tp} a : \texttt{atom} \Rightarrow \Gamma}\ \text{(TP\_A)}$$

$$\frac{}{\Gamma \vdash^{tp} s : \texttt{string} \Rightarrow \Gamma}\ \text{(TP\_S)} \qquad \frac{}{\Gamma \vdash^{tp} b : \texttt{boolean} \Rightarrow \Gamma}\ \text{(TP\_B)}$$

$$\frac{}{\Gamma \vdash^{tp} i : \texttt{integer} \Rightarrow \Gamma}\ \text{(TP\_I)}$$

En estas reglas se expresa que el tipo de los literales es el definido por Elixir, es decir, se utiliza el tipo que el lenguaje le da internamente.

Se define la regla de *subsumption* para patrones de la siguiente forma:

$$\frac{\Gamma \vdash^{tp} p : t \Rightarrow \Gamma^1 \qquad t <| u}{\Gamma \vdash^{tp} p : u \Rightarrow \Gamma^1}\ \text{(TP\_SUB)}$$

Esta regla expresa que un patrón se deriva con tipo u si lo hace para un tipo t que es subtipo de u.

Por ejemplo, el número 1 puede tipar al tipo `float` porque es de tipo `integer` y éste es subtipo de `float`:

$$\frac{}{\Gamma \vdash^{tp} 1 : \texttt{integer} \Rightarrow \Gamma}\ \text{(TP\_I)} \qquad \frac{\texttt{integer} <| \texttt{float}}{\Gamma \vdash^{tp} 1 : \texttt{float} \Rightarrow \Gamma}\ \text{(TP\_SUB)}$$

De esta forma, el número 1 podría ser utilizado en un contexto con tipo `float`:

```
@spec type_of_one(float) :: string

def type_of_one(1.0) do
    "Uno flotante"
end

def type_of_one(1) do
    "Uno entero"
end

def type_of_one(_) do
    "No es uno"
end
```

En el caso de la segunda definición podemos ver como el patrón utilizado es un entero cuando el tipo del parámetro de la función fue especificado como flotante, ya que es un subtipo del mismo:

$$\frac{\begin{array}{c}(\rho.type\_of\_one, 1) \in \Delta \\ \Delta(\rho.type\_of\_one, 1) = \texttt{float} \to \texttt{string} \qquad \Gamma \vdash^{tp} 1 : \texttt{integer} \Rightarrow \Gamma \\ \Delta; \Gamma; \rho \vdash^{t} \text{"}Uno\ entero\text{"} : \texttt{string} \Rightarrow \Gamma \qquad \texttt{integer} <: \texttt{float}\end{array}}{\Delta; \rho \vdash^{ch} \texttt{def}\ type\_of\_one(1)\ \texttt{do}\ \text{"}Uno\ entero\text{"}\ \texttt{end}}\ (\text{CH\_FN})$$

Las siguientes reglas corresponden al tipado de variables:

$$\frac{\begin{array}{c}x \in \Gamma \\ \Gamma(x) = t\end{array}}{\Gamma \vdash^{tp} x : t \Rightarrow \Gamma}\ (\text{TP\_VAR})$$

$$\frac{x \notin \Gamma}{\Gamma \vdash^{tp} x : t \Rightarrow \Gamma[x \mapsto t]}\ (\text{TP\_VARN})$$

La primera regla indica que una variable tiene tipo (t) si se encuentra en el ambiente de variables (Γ) con ese tipo. La segunda indica que si una variable que no se encuentra en el ambiente es utilizada con un determinado tipo, entonces, se agrega al ambiente con ese tipo.

Para expresar que la variable x es agregada al ambiente Γ con un tipo t utilizamos la notación:

$$\Gamma[x \mapsto t] \qquad (\text{N\_ADD\_VAR})$$

Estos son algunos ejemplos de uso de variables:

```
@spec greater_zero(float) :: boolean

def greater_zero(a) do    # a toma el tipo float
    a > 0.0
end

b = 1    # b toma el tipo integer

{b, c} = {1, 2}  # b tenia tipo integer y c toma tipo integer
```

Las siguientes reglas corresponden al tipado de patrones de tipo lista:

$$\frac{}{\Gamma \vdash^{tp} \texttt{[ ]} : \texttt{[}t\texttt{]} \Rightarrow \Gamma}\ (\text{TP\_ELIST})$$

$$\frac{\Gamma \vdash^{tp} p_1 : t \Rightarrow \Gamma^1 \qquad \Gamma^1 \vdash^{tp} p_2 : \texttt{[}t\texttt{]} \Rightarrow \Gamma^2}{\Gamma \vdash^{tp} \texttt{[}p_1 \texttt{|} p_2\texttt{]} : \texttt{[}t\texttt{]} \Rightarrow \Gamma^2}\ (\text{TP\_LIST})$$

La primera regla expresa que las listas vacías tipan a cualquier tipo de listas, es decir, al tipo `[`t`]` sin importar qué tipo sea t. La segunda expresa que un patrón de tipo lista de más de un elemento está bien tipado si el primer elemento de la lista tiene el mismo tipo que los elementos del resto de la lista.

Por ejemplo, la lista `[ 1.2 ]` deriva su tipo de la siguiente forma:

$$\frac{\dfrac{}{\Gamma \vdash^{tp} 1.2 : \texttt{float} \Rightarrow \Gamma}\ (\text{TP\_ELIST}) \qquad \dfrac{}{\Gamma \vdash^{tp} \texttt{[ ]} : \texttt{[float]} \Rightarrow \Gamma}\ (\text{TP\_ELIST})}{\Gamma \vdash^{tp} \texttt{[1.2 | [ ]]} : \texttt{[float]} \Rightarrow \Gamma}\ (\text{TP\_LIST})$$

El siguiente ejemplo muestra cómo se deriva el tipo de un patrón de tipo lista donde el primer elemento es un entero y el resto de los elementos son flotantes:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash^{tp} 1 : \texttt{integer} \Rightarrow \Gamma}\ (\text{TP\_I})}{\Gamma \vdash^{tp} 1 : \texttt{float} \Rightarrow \Gamma}\ (\text{TP\_SUB}) \qquad \dfrac{\vdots}{\Gamma \vdash^{tp} \texttt{[1.2]} : \texttt{[float]} \Rightarrow \Gamma}}{\Gamma \vdash^{tp} \texttt{[1 | [1.2]]} : \texttt{[float]} \Rightarrow \Gamma}\ (\text{TP\_LIST})$$

La siguiente regla corresponde al tipado de un patrón de tipo tupla:

$$\frac{\Gamma \vdash^{tp} p_1 : t_1 \Rightarrow \Gamma^1 \quad \cdots \quad \Gamma^{n-1} \vdash^{tp} p_n : t_n \Rightarrow \Gamma^n}{\Gamma \vdash^{tp} \texttt{\{}p_1\texttt{,}\ldots\texttt{,}p_n\texttt{\}} : \texttt{\{}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{\}} \Rightarrow \Gamma^n}\ (\text{TP\_TUP})$$

En esta regla se expresa que el tipo de un patrón de tipo tupla estará definido por el tipo de cada uno de sus elementos.

Las siguientes reglas corresponden al tipado de patrones de tipo mapa:

$$\frac{}{\Gamma \vdash^{tp} \texttt{\%\{ \}} : \texttt{\%\{ \}} \Rightarrow \Gamma}\ (\text{TP\_EMAP})$$

$$\frac{\Gamma \vdash^{ktp} kp_1 : t \quad \cdots \quad \Gamma \vdash^{ktp} kp_n : t \qquad \Gamma \vdash^{tp} p_1 : u_1 \Rightarrow \Gamma^1 \quad \cdots \quad \Gamma^{n-1} \vdash^{tp} p_n : u_n \Rightarrow \Gamma^n}{\Gamma \vdash^{tp} \texttt{\%\{}kp_1 \Rightarrow p_1\texttt{,}\ldots\texttt{,}kp_n \Rightarrow p_n\texttt{\}} : \texttt{\%\{}t \Rightarrow u_1\texttt{,}\ldots\texttt{,}t \Rightarrow u_n\texttt{\}} \Rightarrow \Gamma^n}\ (\text{TP\_MAP})$$

La regla (TP_MAP) expresa que el tipo de la clave de un patrón de tipo mapa es el mismo para todas las claves.

Por ejemplo, el mapa `%{ 1.0 ⇒ :uno, 2.0 ⇒ true }`, deriva su tipo de la siguiente forma:

$$\frac{\begin{array}{ll}\Gamma \vdash^{ktp} 1.0 : \texttt{float} & \Gamma \vdash^{tp} :\text{uno} : \texttt{atom} \Rightarrow \Gamma^1 \\ \Gamma \vdash^{ktp} 2.0 : \texttt{float} & \Gamma^1 \vdash^{tp} \text{true} : \texttt{boolean} \Rightarrow \Gamma^2\end{array}}{\begin{array}{l}\Gamma \vdash^{tp} \texttt{\%\{}\ 1.0 \Rightarrow :\text{uno}, 2.0 \Rightarrow \text{true}\texttt{\}} : \texttt{\%\{float ⇒ atom,} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{float ⇒ boolean \}} \Rightarrow \Gamma^2\end{array}}\ (\text{TP\_MAP})$$

Notar que para las claves de los patrones se utiliza el juicio ($\vdash^{ktp}$ j. tipado claves) (las reglas correspondientes se detallan más adelante). Además, para simplificar la derivación no se detalla cómo los literales derivan su tipo.

La siguiente regla representa el *binding* para patrones:

$$\frac{\begin{array}{c}\Gamma \vdash^{tp} p_1 : t \Rightarrow \Gamma^1 \\ \Gamma^1 \vdash^{tp} p_2 : t \Rightarrow \Gamma^2\end{array}}{\Gamma \vdash^{tp} p_1 = p_2 : t \Rightarrow \Gamma^2}\ (\text{TP\_BIND})$$

La regla expresa que ambos lados del *binding* deben tener el mismo tipo.

La siguiente regla representa el patrón *wildcard*:

$$\frac{}{\Gamma \vdash^{tp} \_ : t \Rightarrow \Gamma}\ (\text{TP\_WILD})$$

En esta regla se expresa que el patrón *wildcard* tipa con cualquier tipo, por lo que mantiene su semántica de ser utilizado en el lugar de cualquier patrón.

3.3.6.2. Claves de patrones

Las siguientes reglas están relacionadas a las claves de patrones mapas, para definirlas se utilizará el juicio ($\vdash^{ktp}$ j. tipado claves). Estas reglas son muy similares a las de patrones pero tal como indica la sintaxis, no tienen variables, por lo tanto ninguna de ellas modifica el ambiente.

Las siguientes reglas representan las reglas de tipado para los literales:

$$\frac{}{\Gamma \vdash^{ktp} f : \texttt{float}}\ (\text{KTP\_F}) \qquad \frac{}{\Gamma \vdash^{ktp} a : \texttt{atom}}\ (\text{KTP\_A})$$

$$\frac{}{\Gamma \vdash^{ktp} s : \texttt{string}}\ (\text{KTP\_S}) \qquad \frac{}{\Gamma \vdash^{ktp} b : \texttt{boolean}}\ (\text{KTP\_B})$$

$$\frac{(i) = \texttt{integer}}{\Gamma \vdash^{ktp} i : \texttt{integer}}\ (\text{KTP\_I})$$

Se define la regla de *subsumption* para las claves de patrones de la siguiente forma:

$$\frac{\Gamma \vdash^{ktp} p : t \qquad t <| u}{\Gamma \vdash^{ktp} p : u}\ \text{(KTP\_SUB)}$$

Las siguientes reglas corresponden al tipado de patrones claves dados por listas, mapas y tuplas:

$$\frac{}{\Gamma \vdash^{ktp} \texttt{[ ]} : \texttt{[}t\texttt{]}}\ \text{(KTP\_ELIST)}$$

$$\frac{\Gamma \vdash^{ktp} p_1 : t \qquad \Gamma \vdash^{ktp} p_2 : \texttt{[}t\texttt{]}}{\Gamma \vdash^{ktp} \texttt{[}p_1 \texttt{|} p_2\texttt{]} : \texttt{[}t\texttt{]}}\ \text{(KTP\_LIST)}$$

$$\frac{}{\Gamma \vdash^{ktp} \texttt{\%\{ \}} : \texttt{\%\{ \}}}\ \text{(KTP\_EMAP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{ktp} kp_1 : t \\ \cdots \\ \Gamma \vdash^{ktp} kp_n : t \\ \Gamma \vdash^{kltp} p_1 : u_1 \\ \cdots \\ \Gamma \vdash^{ktp} p_n : u_n\end{array}}{\Gamma \vdash^{ktp} \texttt{\%\{}kp_1 \Rightarrow p_1 \ \texttt{,} \ldots \texttt{,} \ kp_n \Rightarrow p_n\texttt{\}} : \texttt{\%\{}t \Rightarrow u_1 \ \texttt{,} \ldots \texttt{,} \ t \Rightarrow u_n\texttt{\}}}\ \text{(KTP\_MAP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{ktp} p_1 : t_1 \\ \cdots \\ \Gamma \vdash^{ktp} p_n : t_n\end{array}}{\Gamma \vdash^{ktp} \texttt{\{}p_1 \texttt{,} \ldots \texttt{,} p_n\texttt{\}} : \texttt{\{}t_1 \texttt{,} \ldots \texttt{,} t_n\texttt{\}}}\ \text{(KTP\_TUP)}$$

La regla para el *binding* es dada por la siguiente regla:

$$\frac{\Gamma \vdash^{ktp} p_1 : t \qquad \Gamma \vdash^{ktp} p_2 : t}{\Gamma \vdash^{ktp} p_1 = p_2 : t}\ \text{(KTP\_BIND)}$$

3.3.6.3. Expresiones

Las reglas de las expresiones estarán definidas por el juicio (j. tipado exp).
Se definen nuevamente las reglas de tipado para los literales:

$$\frac{}{\Delta; \Gamma; \rho \vdash^{t} f : \texttt{float} \Rightarrow \Gamma}\ \text{(TE\_F)} \qquad \frac{}{\Delta; \Gamma; \rho \vdash^{t} a : \texttt{atom} \Rightarrow \Gamma}\ \text{(TE\_A)}$$

$$\frac{}{\Delta; \Gamma; \rho \vdash^{t} s : \texttt{string} \Rightarrow \Gamma}\ \text{(TE\_S)} \qquad \frac{}{\Delta; \Gamma; \rho \vdash^{t} b : \texttt{boolean} \Rightarrow \Gamma}\ \text{(TE\_B)}$$

$$\frac{}{\Delta; \Gamma; \rho \vdash^t i : \texttt{integer} \Rightarrow \Gamma} \text{(TE\_I)}$$

Además, se define la regla de *subsumption* y *downcast*:

$$\frac{\Delta; \Gamma; \rho \vdash^t e : t \Rightarrow \Gamma^1 \quad t <| \, u}{\Delta; \Gamma; \rho \vdash^t e : u \Rightarrow \Gamma^1} \text{(TE\_SUB)}$$

$$\frac{\Delta; \Gamma; \rho \vdash^t e : t \Rightarrow \Gamma^1 \quad u \lll t}{\Delta; \Gamma; \rho \vdash^t e : u \Rightarrow \Gamma^1} \text{(TE\_DOWN)}$$

La siguiente regla expresa el tipado para variables que están definidas en el ambiente:

$$\frac{x \in \Gamma \quad \Gamma(x) = t}{\Delta; \Gamma; \rho \vdash^t x : t \Rightarrow \Gamma} \text{(TE\_VAR)}$$

Esta regla expresa que una variable tiene tipo (t) si se encuentra en el ambiente de variables (Γ) con ese tipo; el ambiente no se modifica. Notar que no es necesario tipar las variables que no se encuentren en el ambiente porque el uso de variables no definidas es controlado por el compilador de Elixir.

El tipado para expresiones de tipo lista se define con las siguientes reglas:

$$\frac{}{\Delta; \Gamma; \rho \vdash^t \texttt{[ ]} : \texttt{[any]} \Rightarrow \Gamma} \text{(TE\_ELIST)}$$

$$\frac{\Delta; \Gamma; \rho \vdash^t e_1 : t \Rightarrow \Gamma^1 \quad \Delta; \Gamma; \rho \vdash^t e_2 : \texttt{[}t\texttt{]} \Rightarrow \Gamma^2}{\Delta; \Gamma; \rho \vdash^t \texttt{[}e_1 \texttt{|} e_2\texttt{]} : \texttt{[}t\texttt{]} \Rightarrow \Gamma^1 \cup_R \Gamma^2} \text{(TE\_LIST)}$$

La regla (TE_LIST) indica que una lista está bien tipada si el primer elemento tiene el mismo tipo que el resto de los elementos.

Se utiliza la siguiente notación para representar la unión entre ambientes:

$$\Gamma^1 \cup_R \Gamma^2 \qquad \text{(unión a la derecha)}$$

Este operador introduce la unión de ambos ambientes, eligiendo los componentes del ambiente de la derecha en caso de coincidencia. Por lo tanto, si ambas expresiones introducen una misma variable con el mismo nombre, permanecerá la introducida por el ambiente de la derecha. Este es el comportamiento en todos los casos de operadores binarios, como se verá en sus correspondientes reglas, y está dada por una decisión de diseño general de Elixir.

Por lo tanto, si se tiene la expresión:

```
(x = 1) === (x = "a")   # false
```

La variable que permanecerá en el ambiente es la x de tipo `string`.

Las siguientes reglas corresponden al tipado de expresiones de tipo mapa y tupla:

$$\frac{}{\Delta; \Gamma; \rho \vdash^t \texttt{\%\{ \}} : \texttt{\%\{ \}} \Rightarrow \Gamma} \text{(TE\_EMAP)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e_1 : t \Rightarrow \Gamma^1 \\ \cdots \\ \Delta; \Gamma; \rho \vdash^t e_n : t \Rightarrow \Gamma^n \\ \Delta; \Gamma; \rho \vdash^t e'_1 : u_1 \Rightarrow \Gamma'^1 \\ \cdots \\ \Delta; \Gamma; \rho \vdash^t e'_n : u_n \Rightarrow \Gamma'^n\end{array}}{\begin{array}{r}\Delta; \Gamma; \rho \vdash^t \texttt{\%\{}e_1 \Rightarrow e'_1, \ldots, e_n \Rightarrow e'_n\texttt{\}} : \texttt{\%\{}t \Rightarrow u_1, \ldots, t \Rightarrow u_n\texttt{\}} \\ \Rightarrow \Gamma^1 \cup_R \Gamma'^1 \cup_R \ldots \cup_R \Gamma^n \cup_R \Gamma'^n\end{array}} \text{(TE\_MAP)}$$

Notar que todos los e_i se tipan bajo el mismo contexto Γ, ya que es el comportamiento que define Elixir. Por ejemplo, el siguiente ejemplo no es válido porque para el segundo elemento del mapa no se conoce la variable a:

```
%{ 1 => (a = 2), 2 => a} # error: 'a' no definida
```

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \cdots \\ \Delta; \Gamma; \rho \vdash^t e_n : t_n \Rightarrow \Gamma^n\end{array}}{\Delta; \Gamma; \rho \vdash^t \texttt{\{}e_1, \ldots, e_n\texttt{\}} : \texttt{\{}t_1, \ldots, t_n\texttt{\}} \Rightarrow \Gamma^1 \cup_R \ldots \cup_R \Gamma^n} \text{(TE\_TUP)}$$

La siguiente regla corresponde al tipado de secuencias de expresiones:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \Delta; \Gamma^1; \rho \vdash^t e_2 : t_2 \Rightarrow \Gamma^2\end{array}}{\Delta; \Gamma; \rho \vdash^t e_1; e_2 : t_2 \Rightarrow \Gamma^2} \text{(TE\_ES)}$$

La secuencia de expresiones tiene el tipo de la expresión de la derecha. La parte izquierda de la secuencia es recorrida con el ambiente de variables inicial. La parte derecha lo hace con el ambiente generado por la parte anterior, extendiendo al ambiente con el que concluye la regla.

Las siguientes reglas aplican para el caso del `if` y el `unless`, agrupados con el símbolo $\odot$:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e : \texttt{boolean} \Rightarrow \Gamma' \\ \Delta; \Gamma'; \rho \vdash^t e_1 : t \Rightarrow \Gamma^1 \\ \Delta; \Gamma'; \rho \vdash^t e_2 : t \Rightarrow \Gamma^2\end{array}}{\Delta; \Gamma; \rho \vdash^t \odot\ e\ \texttt{do}\ e_1\ \texttt{else}\ e_2\ \texttt{end} : t \Rightarrow \Gamma'} \text{(TE\_IF/U\_ELSE)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e : \texttt{boolean} \Rightarrow \Gamma' \\ \Delta;\Gamma';\rho \vdash^t e_1 : t \Rightarrow \Gamma^1\end{array}}{\Delta;\Gamma;\rho \vdash^t \odot\ e\ \texttt{do}\ e_1\ \texttt{end} : t \Rightarrow \Gamma'}\ \text{(TE\_IF/U)}$$

La regla (TE_IF/U_ELSE) aplica cuando se tiene un bloque *else* y la regla (TE_IF/U) cuando no se lo tiene. Las reglas expresan que el tipo de la condición debe ser `boolean`, y tipan al tipo de ambos bloques. Las reglas concluyen con el ambiente generado a partir de tipar la condición. Los bloques no agregan variables fuera de su *scope*, por lo que si se define una variable dentro de un bloque no será válida fuera de éste, comportamiento que se desprende de la semántica de Elixir.

Se utiliza el símbolo $\diamond$ para agrupar las operaciones $\{\texttt{+}, \texttt{-}, \texttt{*}\}$ dado que su chequeo es el mismo. La siguiente regla corresponde al tipado de estas operaciones:

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : t \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : t \Rightarrow \Gamma^2 \\ t <: \texttt{float}\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1 \diamond e_2 : t \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ \text{(TE\_ARITH)}$$

La regla expresa que las expresiones que forman parte de la operación deben tipar a un subtipo de `float`, dado que son operaciones aritméticas. El tipo con el que tipa la regla es el mismo con el que lo hacen los operandos.

Por ejemplo, la operación $1 + 2$ deriva su tipo de la siguiente forma:

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t 1 : \texttt{integer} \Rightarrow \Gamma \\ \Delta;\Gamma;\rho \vdash^t 2 : \texttt{integer} \Rightarrow \Gamma \qquad \texttt{integer} <: \texttt{float}\end{array}}{\Delta;\Gamma;\rho \vdash^t 1 + 2 : \texttt{integer} \Rightarrow \Gamma}\ \text{(TE\_ARITH)}$$

Notar que para simplificar la derivación no se especificó cómo los literales derivan su tipo.

La siguiente regla corresponde al tipado de la división real (`/`):

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : \texttt{float} \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : \texttt{float} \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1 \texttt{/} e_2 : \texttt{float} \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ \text{(TE\_DIV)}$$

La regla es similar a la anterior, pero el tipo de retorno siempre es `float` por ser la división real.

La siguiente regla corresponde al tipado de la negación como operación aritmética (forma de representar a los números negativos):

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e : t \Rightarrow \Gamma^1 \\ t <: \texttt{float}\end{array}}{\Delta;\Gamma;\rho \vdash^t \texttt{-}e : t \Rightarrow \Gamma^1}\ \text{(TE\_NEG)}$$

Esta regla expresa que la expresión a negar debe tener un subtipo del tipo `float`.

Se utiliza el símbolo $\bullet$ para agrupar las operaciones $\{\texttt{and}, \texttt{or}\}$ dado que su chequeo es similar. La siguiente regla corresponde al tipado de estas operaciones:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e_1 : \texttt{boolean} \Rightarrow \Gamma^1 \\ \Delta; \Gamma; \rho \vdash^t e_2 : \texttt{boolean} \Rightarrow \Gamma^2\end{array}}{\Delta; \Gamma; \rho \vdash^t e_1 \bullet e_2 : \texttt{boolean} \Rightarrow \Gamma^1 \cup_R \Gamma^2} \text{(TE\_BOP)}$$

En esta regla se expresa que las expresiones utilizadas para estas operaciones deben tipar con tipo `boolean`, dado que son operadores booleanos, y la regla también tiene tipo `boolean`.

De la misma forma, la siguiente regla representa el tipado de la negación (`not`) para expresiones booleanas:

$$\frac{\Delta; \Gamma; \rho \vdash^t e : \texttt{boolean} \Rightarrow \Gamma^1}{\Delta; \Gamma; \rho \vdash^t \texttt{not}\ e : \texttt{boolean} \Rightarrow \Gamma^1} \text{(TE\_NOT)}$$

Se utiliza el símbolo $\square$ para agrupar las operaciones $\{\texttt{++}, --\}$ dado que su chequeo es igual. La siguiente regla representa el tipado de estas operaciones:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e_1 : [t] \Rightarrow \Gamma^1 \\ \Delta; \Gamma; \rho \vdash^t e_2 : [t] \Rightarrow \Gamma^2\end{array}}{\Delta; \Gamma; \rho \vdash^t e_1 \square e_2 : [t] \Rightarrow \Gamma^1 \cup_R \Gamma^2} \text{(TE\_LOP)}$$

Esta regla expresa que la lista resultante de las operaciones $\square$ tendrá el tipo de las listas con las que se opera.

La siguiente regla representa el tipado de la concatenación de cadenas de caracteres:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e_1 : \texttt{string} \Rightarrow \Gamma^1 \\ \Delta; \Gamma; \rho \vdash^t e_2 : \texttt{string} \Rightarrow \Gamma^2\end{array}}{\Delta; \Gamma; \rho \vdash^t e_1 \texttt{<>} e_2 : \texttt{string} \Rightarrow \Gamma^1 \cup_R \Gamma^2} \text{(TE\_CONCAT)}$$

Las cadenas a concatenar deben tipar a un subtipo del tipo `string`, y la regla tipa siempre como `string`.

La siguiente regla representa el tipado de la obtención de un elemento de un mapa:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e_1 : \texttt{\%\{}t \Rightarrow u_1\texttt{,...,}t \Rightarrow u_n\texttt{\}} \Rightarrow \Gamma^1 \\ \Delta; \Gamma; \rho \vdash^t e_2 : t \Rightarrow \Gamma^2\end{array}}{\Delta; \Gamma; \rho \vdash^t e_1\,[e_2] : \texttt{any} \Rightarrow \Gamma^1 \cup_R \Gamma^2} \text{(TE\_MAPAPP)}$$

La clave (e_2) para la cual se quiere obtener el valor debe ser del tipo al que tipan las claves del mapa (e_1). La regla siempre tiene tipo `any` porque no se puede asegurar la existencia de la clave en el mapa, ni el tipo que tiene el valor para esa clave, por lo tanto la expresión queda "no tipada". Esta es una decisión de diseño que se tomó, pero se podrían tomar otros enfoques como por ejemplo, solo permitir la aplicación de literales para ser capaz de determinar estáticamente el campo seleccionado y consecuentemente el tipo del valor, tal como se propone en (Cassola y cols., 2020).

Para agrupar las operaciones de comparación {`==`, `!=`, `===`, `!==`, `>`, `<`, `>=`, `<=`} utilizaremos el símbolo $\star$. La siguiente regla representa el tipado de estas operaciones:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \Delta; \Gamma; \rho \vdash^t e_2 : t_2 \Rightarrow \Gamma^2\end{array}}{\Delta; \Gamma; \rho \vdash^t e_1 \star e_2 : \texttt{boolean} \Rightarrow \Gamma^1 \cup_R \Gamma^2} \text{(TE\_CMP)}$$

Como se puede observar, se permite comparar expresiones que tengan distintos tipos. Esto se debe a que Elixir internamente establece un orden entre los diferentes tipos, y permite dicha comparación. La regla tipa con tipo `boolean`.

La siguiente regla corresponde al tipado de una sentencia condicional *case*:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e : t \Rightarrow \Gamma' \\ \varnothing \vdash^{tp} p_1 : t \Rightarrow \Gamma^1 \\ \cdots \\ \varnothing \vdash^{tp} p_n : t \Rightarrow \Gamma^2 \\ \Delta; \Gamma' \cup_R \Gamma^1; \rho \vdash^t e_1 : t' \Rightarrow \Gamma'^1 \\ \cdots \\ \Delta; \Gamma' \cup_R \Gamma^n; \rho \vdash^t e_n : t' \Rightarrow \Gamma'^n\end{array}}{\Delta; \Gamma; \rho \vdash^t \texttt{case}\ e\ \texttt{do}\ p_1 \to e_1, ..., p_n \to e_n\ \texttt{end} : t' \Rightarrow \Gamma'} \text{(TE\_CASE)}$$

Esta regla expresa que cada patrón de las guardas debe tipar con el mismo tipo que el selector e. Los cuerpos de cada rama son tipados con el ambiente generado al tipar el selector, extendido por las variables del patrón de la guarda correspondiente. Los cuerpos de las ramas deben tipar al mismo tipo y ese es el tipo con el que tipa la regla. Notar que los bloques generan un ambiente que solo tendrán alcance dentro de sí mismos. El ambiente resultante de la regla es el obtenido de tipar el selector (e).

La siguiente regla corresponde al tipado de una sentencia condicional *cond*:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t e_1 : \texttt{boolean} \Rightarrow \Gamma^1 \\ \cdots \\ \Delta; \Gamma; \rho \vdash^t e_n : \texttt{boolean} \Rightarrow \Gamma^n \\ \Delta; \Gamma^1; \rho \vdash^t e'_1 : t \Rightarrow \Gamma^{1'} \\ \cdots \\ \Delta; \Gamma^n; \rho \vdash^t e'_n : t \Rightarrow \Gamma^{n'}\end{array}}{\Delta; \Gamma; \rho \vdash^t \texttt{cond do}\ e_1 \to e'_1, ..., e_n \to e'_n\ \texttt{end} : t \Rightarrow \Gamma} \text{(TE\_COND)}$$

Cada condición debe tipar con tipo `boolean`. Los cuerpos de cada guarda son tipados con el ambiente obtenido de tipar la guarda correspondiente. Las expresiones de las ramas deben tipar al mismo tipo y ese es el tipo con el que tipa la regla. Es necesario destacar que el ambiente inicial no es modificado.

La siguiente regla corresponde al tipado del *binding* de expresiones:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^{t} e : t \Rightarrow \Gamma^{1} \\ \varnothing \vdash^{tp} p : t \Rightarrow \Gamma^{2}\end{array}}{\Delta; \Gamma; \rho \vdash^{t} p = e : t \Rightarrow \Gamma^{1} \cup_{R} \Gamma^{2}} \text{(TE\_BIND)}$$

La regla tiene el tipo de la expresión de la derecha. El siguiente ejemplo muestra cómo la expresión $x = :foo$ deriva su tipo:

$$\frac{\Delta; \Gamma; \rho \vdash^{t} :foo : \texttt{atom} \Rightarrow \Gamma \qquad \dfrac{x \notin \varnothing}{\varnothing \vdash^{tp} x : \texttt{atom} \Rightarrow \{(x, \texttt{atom})\}}\ \text{(TP\_VARN)}}{\Delta; \varnothing; \rho \vdash^{t} x = :foo : \texttt{atom} \Rightarrow \{(x, \texttt{atom})\}}\ \text{(TE\_BIND)}$$

Notar que para simplificar la derivación no se especificó cómo el átomo deriva su tipo y se utilizó el conjunto vacío como el ambiente inicial.

La siguiente regla corresponde al tipado de la llamada a una función que se encuentra en el ambiente de funciones, con un prefijo:

$$\frac{\begin{array}{c}\Delta(\rho_2.f\_name, n) = (t_1, \ldots, t_n) \to t \\ \Delta; \Gamma; \rho_1 \vdash^{t} e_1 : t'_1 \Rightarrow \Gamma^{1} \\ \ldots \\ \Delta; \Gamma; \rho_1 \vdash^{t} e_n : t'_n \Rightarrow \Gamma^{n} \\ t'_1 <: t_1 \quad \ldots \quad t'_n <: t_n\end{array}}{\Delta; \Gamma; \rho_1 \vdash^{t} \rho_2.f\_name\ (e_1, ..., e_n) : t \Rightarrow \Gamma^{1} \cup_{R} ... \cup_{R} \Gamma^{n}} \text{(TE\_APPFE)}$$

Para este caso la función se busca en el ambiente con el prefijo especificado en el llamado. Cada expresión usada como parámetro debe tipar a un subtipo del tipo con el que está definido ese parámetro en el ambiente. Notar que se utiliza el subtipado ($<:$) porque las funciones pueden definir sus parámetros como `any`, y la regla de (TE_SUB) no es suficiente para este caso. Definir un parámetro como `any` brinda la flexibilidad para no tipar (o tipar parcialmente) ese parámetro, y que el sistema de tipos no falle. El tipo con el que tipa la regla es el tipo de retorno con el que está definida la función en el ambiente.

El siguiente ejemplo es una derivación de una llamada a una función que existe en el ambiente:

$$\frac{\begin{array}{c}\Delta(M.func, 1) = \texttt{float} \to \texttt{string} \\ \Delta; \Gamma; \rho_1 \vdash^{t} 1 : \texttt{integer} \Rightarrow \Gamma \\ \texttt{integer} <: \texttt{float}\end{array}}{\Delta; \Gamma; \rho_1 \vdash^{t} M.func(1) : \texttt{string} \Rightarrow \Gamma} \text{(TE\_APPFE)}$$

Notar que para simplificar la derivación no se especificó cómo el literal 1 deriva con tipo `integer`.

La siguiente regla corresponde al tipado de la llamada a una función que no se encuentra en el ambiente de funciones, con un prefijo:

$$\frac{\begin{array}{c}(\rho_2.f\_name, n) \notin \Delta \\ \Delta; \Gamma; \rho_1 \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \cdots \\ \Delta; \Gamma; \rho_1 \vdash^t e_n : t_n \Rightarrow \Gamma^n\end{array}}{\Delta; \Gamma \rho_1 \vdash^t \rho_2.f\_name\ (e_1, ..., e_n) : \texttt{any} \Rightarrow \Gamma^1 \cup_R ... \cup_R \Gamma^n}\ \text{(TE\_NAPPFE)}$$

Esta regla expresa que el llamado a una función que no se encuentra en el ambiente tiene tipo `any` y no modifica el ambiente de variables.

El siguiente ejemplo es una derivación de una llamada a una función que no existe en el ambiente:

$$\frac{\begin{array}{c}(Math.sqrt, 1) \notin \Delta \\ \Delta; \Gamma; \rho_1 \vdash^t 4 : \texttt{integer} \Rightarrow \Gamma\end{array}}{\Delta; \Gamma; \rho_1 \vdash^t Math.sqrt(4) : \texttt{any} \Rightarrow \Gamma}\ \text{(TE\_NAPPFE)}$$

Notar que para simplificar la derivación no se especificó cómo el literal 4 deriva con tipo `integer`.

Para poder utilizar estas funciones que tipan al tipo `any` se utiliza la regla TE_DOWN que reduce este tipo a otros más específicos. El siguiente ejemplo muestra como una función externa puede definir su tipo como `integer` mediante la regla (TE_DOWN):

$$\frac{\dfrac{\begin{array}{c}(Math.sqrt, 1) \notin \Delta \\ \Delta; \Gamma; \rho_1 \vdash^t 4 : \texttt{integer} \Rightarrow \Gamma\end{array}}{\Delta; \Gamma; \rho_1 \vdash^t Math.sqrt(4) : \texttt{any} \Rightarrow \Gamma}\ \text{(TE\_NAPPFE)} \qquad \texttt{integer} \lll \texttt{any}}{\Delta; \Gamma; \rho \vdash^t Math.sqrt(4) : \texttt{integer} \Rightarrow \Gamma}\ \text{(TE\_DOWN)}$$

El uso de estas funciones, o especificar el tipo de retorno como `any` para una función como se explicará más adelante con ejemplos, debe ser responsabilidad del desarrollador.

Un ejemplo de uso válido es el siguiente:

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^t 1 : \texttt{integer} \Rightarrow \Gamma \\ \Delta; \Gamma; \rho \vdash^t Math.sqrt(4) : \texttt{integer} \Rightarrow \Gamma \qquad \texttt{integer} <: \texttt{float}\end{array}}{\Delta; \Gamma; \rho \vdash^t 1 + Math.sqrt(4) : \texttt{integer} \Rightarrow \Gamma}\ \text{(TE\_ARITH)}$$

Notar que no se especificó como derivan sus tipos las expresiones para simplificar.

Por otro lado, el siguiente uso de la función se deriva de forma válida, sin embargo falla en tiempo de ejecución:

$$\frac{\Delta;\Gamma;\rho \vdash^t \texttt{"}hola\texttt{"}: \texttt{string} \Rightarrow \Gamma \quad \Delta;\Gamma;\rho \vdash^t Math.sqrt(4) : \texttt{string} \Rightarrow \Gamma}{\Delta;\Gamma;\rho \vdash^t \texttt{"}hola\texttt{"}\texttt{<>}Math.sqrt(4) : \texttt{string} \Rightarrow \Gamma}\ \text{(TE\_CONCAT)}$$

Donde la función externa $Math.sqrt(4)$ deriva con tipo `string` de la siguiente forma:

$$\frac{\dfrac{(Math.sqrt, 1) \notin \Delta \quad \Delta;\Gamma;\rho_1 \vdash^t 4 : \texttt{integer} \Rightarrow \Gamma}{\Delta;\Gamma;\rho_1 \vdash^t Math.sqrt(4) : \texttt{any} \Rightarrow \Gamma}\ \text{(TE\_NAPPFE)} \qquad \texttt{string} \lll \texttt{any}}{\Delta;\Gamma;\rho \vdash^t Math.sqrt(4) : \texttt{string} \Rightarrow \Gamma}\ \text{(TE\_DOWN)}$$

En los ejemplos anteriores, se ve claramente la gradualidad del sistema de tipos ya que sólo los fragmentos con tipos especificados se chequean, y en otros casos el tipo `any` se puede castear a cualquier otro, teniendo entonces un chequeo dinámico. Para estos casos, el tipo `any` no se usa como súper tipo, si no como el tipo desconocido del *gradual typing*.

Las siguientes reglas representan el tipado de las llamadas a funciones sin un prefijo:

$$\frac{\begin{array}{c}\Delta(\rho.f\_name, n) = (t_1, \ldots, t_n) \to t \\ \Delta;\Gamma;\rho \vdash^t e_1 : t'_1 \Rightarrow \Gamma^1 \\ \cdots \\ \Delta;\Gamma;\rho \vdash^t e_n : t'_n \Rightarrow \Gamma^n \\ t'_1 <: t_1 \quad \ldots \quad t'_n <: t_n\end{array}}{\Delta;\Gamma;\rho \vdash^t f\_name\ (e_1, ..., e_n) : t \Rightarrow \Gamma^1 \cup_R ... \cup_R \Gamma^n}\ \text{(TE\_APPFI)}$$

$$\frac{\begin{array}{c}(\rho.f\_name, n) \notin \Delta \\ \Delta;\Gamma;\rho \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \cdots \\ \Delta;\Gamma;\rho \vdash^t e_n : t_n \Rightarrow \Gamma^n\end{array}}{\Delta;\Gamma\rho \vdash^t f\_name\ (e_1, ..., e_n) : \texttt{any} \Rightarrow \Gamma^1 \cup_R ... \cup_R \Gamma^n}\ \text{(TE\_NAPPFI)}$$

Estas últimas dos reglas son iguales a las dos anteriores, pero sin prefijo, por lo que la función se busca en el ambiente con el prefijo inicial (ρ).

3.3.7. Pseudopolimorfismo

En esta sección se describirá a través de ejemplos como se logra una aproximación a polimorfismo paramétrico para el sistema de tipos que se presenta.

Para definir una función que pueda ser aplicada para todos los tipos se debe anotar a los tipos de los parámetros como `any`. De esta forma cualquiera sea el tipo del parámetro con el que se llame siempre será subtipo de éste, ya que en la jerarquía de tipos `any` es el *top type*.

El siguiente ejemplo muestra una función polimórfica que retorna verdadero si el elemento que se pasa es igual a si mismo:

```
@spec true(any) :: boolean
def true(x) do
    x == x
end
```

Algunos ejemplos de invocación a la función anterior son:

```
true([]) # true
true(1) # true
true("hola") # true
true(%{:a => "a", :b => b}) # true
```

Otro ejemplo puede ser una función polimórfica que suma uno al parámetro que se le pasa:

```
@spec add_1(any) :: integer
def add_1(x) do
    x + 1
end
```

Algunos ejemplos de invocación a la función anterior son:

```
add_1(1) # 2
add_1(40 * 2) # 81

add_1("hola") # error
```

Aquí nuevamente se ve la gradualidad del sistema de tipos ya que ese error no lo detecta, y falla en tiempo de ejecución.

Cuando se quiere especificar una función para una lista de un determinado tipo (enteros por ejemplo), se puede hacer de la siguiente forma:

```
@spec length([integer]) :: integer
def length([]) do
    0
end
def length([head|tail]) do
    1 + length(tail)
end
```

Algunos ejemplos de invocación a la función anterior son:

```
length([]) # 0
length([1, 2, 3]) # 3

length(["1", "2", "3"]) # error
length([:uno, :dos, :tres]) # error
length([1, :dos, "tres"]) # error
```

Acá se puede ver como la lista vacía no da error por ser del tipo `[any]` y puede ser reducida al tipo `[integer]` mediante la regla (TE_DOWN).

Pero, si se quiere definir una función que sea aplicable a todas las listas, se debe indicar con el tipo `[any]`.

La siguiente es una función que toma una lista y retorna su largo sin importar el tipo de sus valores:

```
@spec length_any([any]) :: integer
def length_any([]) do
    0
end
def length_any([head|tail]) do
    1 + length_any(tail)
end
```

Algunos ejemplos de invocación a la función anterior son:

```
length_any([]) # 0
length_any([1, 2, 3]) # 3
length_any(["1", "2", "3"]) # 3
length_any([:uno, :dos, :tres]) # 3

length_any([1, :dos, "tres"]) # error
```

Las listas deben ser todas de un mismo tipo, es por esto que el último caso es erróneo. Notar que este error se da en tiempo de compilación ya que se chequea en el llamado de la función.

Para el caso de los mapas, si se quiere especificar una función sobre mapas donde sus claves sean átomos se puede hacer de la siguiente forma:

```
@spec key_is_uno(%{atom => any}) :: boolean
def key_is_uno(map) do
    map[:key1] == :uno
end
```

Los mapas tienen un funcionamiento particular porque las claves siempre deben ser de un mismo tipo, por lo tanto una vez que se determina, todas las claves deben cumplirlo (mismo caso que las listas). Sin embargo, los valores pueden ser de tipos diferentes. Un mapa con más valores es subtipo de uno con menos, siempre y cuando se cumplan los tipos que sí están definidos para la clave y los valores. En este caso se especificó el tipo de los valores como any, por lo tanto cualquier tipo de mapa que cumpla el tipo de la clave será subtipo de éste tipo.

Ejemplos de invocación a la función anterior:

```
key_is_uno(%{:key1=>1, :key2=>:dos, :key3=>"tres"}) # false
key_is_uno(%{:key1=>:uno, :key2=>:dos, :key3=>"tres"}) # true

key_is_uno(%{"1"=>:dos, "dos"=>:dos}) # error porque las
    claves no son atomos
```

```
key_is_uno(%{:key1=>:uno, "dos"=>:dos, 3=>:tres}) # error
    porque las claves son de distintos tipos
```

Si se quiere especificar una función que tome como parámetro mapas con cualquier tipo en su clave se debe indicar que éstas son de tipo `none`. La siguiente función toma un mapa polimórfico y retorna verdadero si para la clave *:key1* se tiene como valor el átomo *:uno*:

```
@spec key_is_uno_pol(%{none => any}) :: boolean
def key_is_uno_pol(map) do
    map[:key1] == :uno
end
```

Ejemplos de invocación a la función anterior son:

```
key_is_uno_pol(%{"uno"=>:uno, "dos"=>2, "tres"=>"Tres"}) #
    false
key_is_uno_pol(%{"uno"=>1, "dos"=>2, "tres"=>3}) # false
key_is_uno_pol(%{:key1=>:uno, :key2=>2, :key3=>"Tres"}) #
    true
key_is_uno_pol(%{:key1=>:one, :key2=>:two, :key3=>:three}) #
    false

key_is_uno_pol(%{1=>:uno, :dos=>2, "tres"=>"Tres"}) # error
    porque las claves son de distintos tipos
```

Es necesario recordar que para el subtipado los mapas son covariantes en los tipos de los valores y contravariantes en los tipos de las claves. Un ejemplo de la relación de subtipado entre el tipo del mapa de la primer llamada con el tipo del parámetro de la función sería:

$$\frac{\texttt{none} <: \texttt{string} \qquad \texttt{atom} <: \texttt{any}}{\texttt{\%\{string} \Rightarrow \texttt{atom, string} \Rightarrow \texttt{integer, string} \Rightarrow \texttt{string\}} <: \texttt{\%\{none} \Rightarrow \texttt{any\}}} \text{(ST\_MAP)}$$

Entonces, siendo $e = \texttt{\%\{}\text{“}uno\text{”} \Rightarrow \texttt{:}uno\texttt{,}\text{“}dos\text{”} \Rightarrow 2 \texttt{ ,}\text{“}tres\text{”} \Rightarrow \text{“}Tres\text{”}\texttt{\}}$, la regla de tipado para la llamada sería:

$$\frac{\begin{array}{c}\Delta(\rho_1.key\_is\_uno\_pol, 1) = \texttt{\%\{none} \Rightarrow \texttt{any\}} \rightarrow \texttt{boolean} \\ \Delta; \Gamma; \rho_1 \vdash^t e : \texttt{\%\{string} \Rightarrow \texttt{atom, string} \Rightarrow \texttt{integer, string} \Rightarrow \texttt{string\}} \Rightarrow \Gamma \\ \texttt{\%\{string} \Rightarrow \texttt{atom, string} \Rightarrow \texttt{integer, string} \Rightarrow \texttt{string\}} <: \texttt{\%\{none} \Rightarrow \texttt{any\}}\end{array}}{\Delta; \Gamma; \rho_1 \vdash^t key\_is\_uno\_pol\ (e) : \texttt{boolean} \Rightarrow \Gamma} \text{(TE\_APPFE)}$$

Si se quiere definir una función que recibe una tupla donde uno de sus elementos sea cualquier tipo se debe especificar el tipo de ese valor como `any`.

La siguiente función toma duplas donde el primer elemento puede ser de cualquier tipo mientras que el segundo debe ser un entero:

```
@spec second_gt_2({any, integer}) :: boolean
def second_gt_2({x, y}) do
    y > 2
end
```

Ejemplos de invocación a la función anterior:

```
second_gt_2({1, 1}) # false
second_gt_2({2, 3}) # true
second_gt_2({:dos, 3}) # true
second_gt_2({"uno", 4}) # true

second_gt_2({5,"tres"}) # error
```

Para no definir un tipo específico de retorno de una función se puede anotarlo como `any`.

Por ejemplo, la siguiente función toma una lista de cualquier tipo y retorna su primer elemento:

```
@spec head([any]) :: any
def head(list) do
    [head | tail] = list
    head
end
```

Ejemplos de invocación a la función anterior:

```
head(["uno", "dos", "tres"]) # "uno"
head([1, 2, 3]) # 1
head([:uno, :dos, :tres]) # :uno
head([[1,2,3], [4,5,6], [7,8,9]]) # [1,2,3]
```

De la misma forma que lo hicimos para los parámetros, también podemos decir que el tipo de retorno será una lista pero de cualquier tipo, anotando el tipo de retorno como `[any]`. El siguiente ejemplo muestra la especificación de una función que tiene como parámetro una lista de cualquier tipo y como retorno una lista (la cola) de la cual no se conoce su tipo:

```
@spec tail([any]) :: [any]
def tail(list) do
    [head | tail] = list
    tail
end
```

Ejemplos de invocación a la función anterior:

```
tail([1]) # []
tail([1,2]) # [2]
tail([1.1, 2.0]) # [2.0]
tail(["uno", "dos", "tres"]) # ["dos", "tres"]
tail([:uno, :dos]) # [:dos]
```

```
tail([{1,"uno"}, {2,"dos"}, {3,"tres"}]) # [{2,"dos"}, {3,"
    tres"}]
tail([%{1 => 3}, %{2 => "4"}, %{3 => :cinco}]) # [%{2 =>
    "4"}, %{3 => :cinco}]
```

Como se mencionó, al brindarse esta flexibilidad mediante la regla (TE_DOWN), es responsabilidad del desarrollador el uso de este tipo de funciones ya que por ejemplo, podríamos tener los siguientes usos:

```
length(tail([0,1]))
length(tail(["aa", "bb"]))
```

Ambos ejemplos son válidos en tiempo de ejecución porque la función *length* requiere una lista de enteros y la función *tail* retorna una lista de valores `any`. Por downcast se tiene:

$$\frac{\Delta;\Gamma;\rho \vdash^t tail([0,1]) : \texttt{[any]} \Rightarrow \Gamma^1 \qquad \texttt{[integer]} \lll \texttt{[any]}}{\Delta;\Gamma;\rho \vdash^t tail([0,1]) : \texttt{[integer]} \Rightarrow \Gamma^1}\ \text{(TE\_DOWN)}$$

$$\frac{\Delta;\Gamma;\rho \vdash^t tail([\ "aa",\ "bb"]) : \texttt{[any]} \Rightarrow \Gamma \qquad \texttt{[integer]} \lll \texttt{[any]}}{\Delta;\Gamma;\rho \vdash^t tail([\ "aa",\ "bb"]) : \texttt{[integer]} \Rightarrow \Gamma}\ \text{(TE\_DOWN)}$$

Sin embargo, en tiempo de ejecución el segundo caso dará un error, por utilizar una lista de cadena de caracteres en una función que esperaba una lista de enteros.

De la misma forma se puede obtener este comportamiento para mapas y tuplas.

Capítulo 4

Experimentación

En este capítulo se presentan varios tipos de análisis con diferentes enfoques. El primero de ellos tiene por cometido presentar diferentes pruebas sobre el sistema de tipos. El segundo es un análisis de la implementación realizada del chequeador de tipos, luego de publicada la primera versión de la biblioteca. Por último, se detallan los resultados de entrevistar a José Valim, creador de Elixir, y la aceptación y publicación de un artículo basado en el sistema de tipos para la *SBLP 2020: 24th Brazilian Symposium on Programming Languages*.

4.1. Análisis del sistema de tipos

En el capítulo anterior se introdujeron, a medida que se detallaban las reglas, algunos ejemplos de derivaciones para demostrar el uso de las reglas. Éstas han servido como forma de demostrar la consistencia del sistema de tipos y cómo son derivados algunos casos de prueba. En el Apéndice E se detallan más derivaciones para otros casos más complejos. Aunque no se realiza una prueba formal sobre el sistema de tipos, como robustez o preservación de tipos, estos ejemplos permiten mostrar que los casos de uso más frecuentes pueden ser derivados de forma correcta.

Por otro lado, la biblioteca desarrollada está basada en el sistema de tipos, por lo tanto, se demuestra que se puede realizar un chequeador de tipos basado en dicho sistema. Los casos de prueba que se realizan sobre la biblioteca también, indirectamente, están probando el sistema de tipos.

4.2. Creación de biblioteca a partir del sistema de tipos

Existen diferentes alternativas y opciones para implementar la solución, como puede ser realizarlo a nivel de lexer y parser dentro del compilador (y así trabajar con el AST que genera Elixir), modificar el código en sí del compilador

de Elixir (dado que es un lenguaje *open source*), meta programar y chequear una sintaxis nueva o implementar una tarea de Mix.

Se optó por implementar una tarea de Mix con la motivación de no modificar el lenguaje, y realizar la implementación de una biblioteca la cual se pueda añadir como dependencia a un proyecto Elixir (Typelixir, 2020). El desarrollador será capaz de ejecutarla cuando quiera mediante el comando ***mix typelixir***, conservando las demás funcionalidades del compilador. Por lo tanto, se tendrá disponible la compilación estándar (***mix compile***), y la nueva compilación gradual.

Es una solución que no está incluida en el proceso de compilación estándar pero es realizada de forma posterior a este. De esta forma, el chequeo de tipos se hace sobre código Elixir válido. Este código válido implica, por ejemplo, que se realizó un chequeo de la sintaxis, que no existe ambigüedad en las invocaciones a funciones, e incluso, que no existen ciclos en las importaciones de módulos.

Se trabaja con el AST extendido de Elixir, por lo tanto si se observa el flujo de compilación, se está agregando una fase luego de la expansión del AST y antes de la traducción al AST de Erlang. La figura 4.1 refleja esto y muestra todo el proceso que se realiza cuando se ejecuta la tarea:

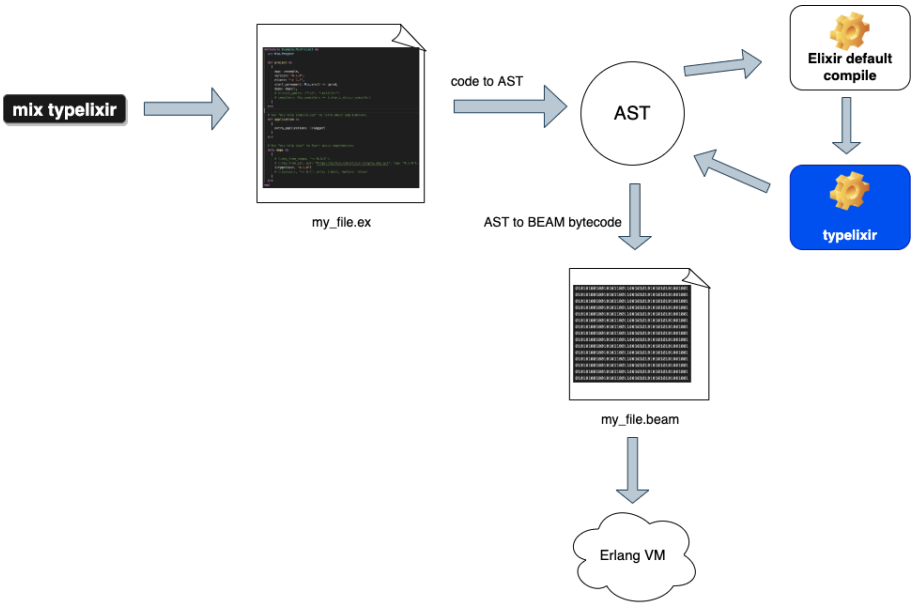


Figura 4.1: Proceso de compilación

Los tipos internos de Elixir se mapean con los utilizados en el sistema de tipos. No se hace ninguna modificación en la sintaxis del lenguaje, incluso la directiva *@spec* es propia de Elixir. Además, se brinda el soporte necesario para anotar a las funciones con los tipos `any` y `none` (este último para el caso de la clave de los mapas).

En primer lugar se realiza la recorrida por los diferentes archivos recolectando las funciones de los módulos, siguiendo las reglas de recorrida que se presentaron. Luego, el módulo principal de la biblioteca es el encargado de realizar una nueva recorrida del árbol en pre-orden, definiendo una función `process` la cual mediante el pattern matching de las diferentes partes del árbol va realizando los chequeos que se presentaron en el sistema de tipos, y en caso de error, se aborta la recorrida para mostrar el mensaje en consola. Por lo tanto, solo se muestra un único error de tipos cada vez que se realiza el chequeo.

En la figura 4.2 se puede ver un ejemplo de un programa, el correspondiente AST generado por Elixir y la salida de ejecutar la tarea:

```elixir
defmodule Example do
  @spec is_1(integer) :: boolean
  def is_1(x) do
    1
  end
end
```

```
{:ok,
 {:defmodule, [line: 1],
  [
   {:__aliases__, [line: 1], [:Example]},
   [
    do: {:__block__, [],
     [
      {:@, [line: 2],
       [
        {:spec, [line: 2],
         [
          {:"::", [line: 2],
           [
            {:is_1, [line: 2], [{:integer, [line: 2], nil}]},
            {:boolean, [line: 2], nil}
           ]
          }
         ]
        }
       ]
      },
      {:def, [line: 3],
      [{:is_1, [line: 3], [{:x, [line: 3], nil}]}, [do: 1]]}
     ]
    }
   ]
  ]
 }
}
```

```
Typelixir -> Compiling 1 file (.ex)

error: Body doesn't match function type on is_1/1 declaration
  lib/example.ex:3
```

Figura 4.2: AST generado por Elixir

Por último, es importante mencionar que los mensajes de error que muestra el analizador están diseñados para que sean útiles para el programador. Se busca que los errores no sean ambiguos, que contengan la información necesaria para encontrar el error y poder solucionarlo, y que sean amigables. Algunos ejemplos de mensajes de error se encuentran en las Figuras 4.3 y 4.4:

```
1  defmodule Example do
2    def test(x) do
3      a = %{}
4      b = %{1 => "a"}
5      c = %{40 => :value, 47.5 => :o_value, 30 => length{[1]}}
6      d = %{1 => 2, "2" => 3}
7    end
8  end
9
```

```
Typelixir -> Compiling 1 file (.ex)

error: Malformed type map
  lib/example.ex:6
```

Figura 4.3: Mapa mal formado

```
 1  defmodule Example do
 2    def test() do
 3      a = 1 + 2
 4      b = 2 + 3.4
 5      c = 1 / 2
 6      d = length([]) + 2
 7      e = 4 + "5"
 8    end
 9  end
10
```

```
Typelixir -> Compiling 1 file (.ex)

error: Type error on + operator
  lib/example.ex:7
```

Figura 4.4: Operador aritmético erróneo

En el Apéndice C, se pueden ver más ejemplos de éstos.

4.3. Análisis de la biblioteca

Sobre la biblioteca se realizaron distintos tipos de análisis. Por un lado, se realizaron pruebas unitarias en el código implementado, chequeando que cada función definida en la biblioteca funciona correctamente por separado. Por ejemplo, se definió una función de subtipado que implementa la relación presentada en el sistema de tipos, y se escribieron pruebas unitarias sobre dicha función, chequeando el correcto funcionamiento de todos los casos.

Por otro lado, se realizó, junto con los tutores, pruebas funcionales exhaustivas sobre la biblioteca. De esta forma se testeó como la biblioteca funciona correctamente para los casos de prueba que se definieron. Además, se utilizaron módulos de proyectos existentes para probarla sobre casos de uso reales. Algunos ejemplos pueden verse en el Apéndice C.

A partir de los resultados obtenidos, se puede determinar que la biblioteca es aplicable a casos de uso reales, utilizados hoy en día en la industria. De todas maneras, tiene la limitante de que solo está definida para un conjunto reducido de operaciones, y por lo tanto, no es aplicable a todos los proyectos. Para aquellas operaciones que sí están definidas, el analizador se comporta como es esperado, brindando mensajes de error claros.

4.4. Entrevista a José Valim y Presentación en SBLP 2020

Una vez finalizado el sistema de tipos se optó por realizarle una entrevista por mail a José Valim, el creador de Elixir, con el fin de compartir el enfoque que toma la biblioteca con alguien que conoce el lenguaje en detalle y así conocer sus pensamientos y recomendaciones para trabajo a futuro.

La respuesta de José, detallada en el Apéndice D, es un indicador de que el rumbo que toma nuestro enfoque está alineado con los objetivos del equipo principal de Elixir. Actualmente, ellos se encuentran trabajando en un sistema de tipos basado en *gradual typing*, aunque decidieron basarse en inferencia de tipos. A diferencia de nuestra solución, que es una biblioteca externa, ellos buscan que este chequeo sea parte del lenguaje.

En cuanto a trabajo a futuro, destacó que una parte muy interesante para continuar es explorar la posibilidad de aplicar *behaviours* y *protocols* en un contexto de *gradual typing*.

Por otro lado, en conjunto con los tutores, el sistema de tipos fue presentado en forma de artículo para la *SBLP 2020: 24th Brazilian Symposium on Programming Languages*, el cual fue aceptado y presentado en la conferencia los días 19 a 23 de octubre de 2020 (Cassola y cols., 2020).

Aunque se tuvieron que hacer algunas modificaciones a pedido de los revisores, la aceptación del paper en la conferencia es un indicador de que la temática es de interés tanto en la academia como en la industria. Por un lado, porque es un lenguaje relativamente nuevo y no existen investigaciones extensivas sobre dicho lenguaje y por otro, porque la temática sobre sistemas de tipos para los

lenguajes de programación es un problema de estudio conocido y de interés, y nuestra solución contribuye a este estudio.

Capítulo 5

Conclusiones

Hemos introducido un sistema de tipos gradual para un fragmento de Elixir, que establece un compromiso entre la seguridad del tipado estático y la flexibilidad del tipado dinámico.

La mayoría de los enfoques previos para el tipado gradual (Siek y Taha, 2006, 2007) se basan en una relación simétrica llamada *consistencia*. Nuestro enfoque es más cercano al de (Castagna y cols., 2019), que se basa en una relación de materialización (precisión) y subtipado. Nos diferenciamos de esto en que usamos el *top type* `any` (súper tipo de la relación de subtipo) como el tipo desconocido del *gradual typing*.

Con este trabajo se genera otro precedente relacionado a los sistemas de tipos en lenguajes de programación en el contexto de proyecto de grado, y en particular para un lenguaje relativamente nuevo y utilizado activamente en la industria hoy en día. Se logra introducir el tema de tipado estático y dinámico con un nivel de profundidad extensivo, y se genera material que puede ser de utilidad para futuros trabajos dentro del mismo contexto. A pesar de ser este un tema con varios años de estudio, su aplicación a Elixir es algo relativamente reciente y los trabajos disponibles no son abundantes como en otros lenguajes. Luego de intensas búsquedas se logró recolectar algunos trabajos que en su globalidad buscan alcanzar el mismo objetivo.

Se introduce una biblioteca creada en el propio lenguaje que realiza un chequeo de tipos acorde al detallado en el sistema de tipos presentado.

A lo largo de la elaboración de este proyecto, se rescatan muchas lecciones aprendidas, que se detallan en los párrafos siguientes.

Definir la sintaxis para un lenguaje es una tarea compleja. Aunque Elixir es de código abierto y cuenta con buena documentación, definir formalmente un conjunto consistente de su sintaxis necesita de muchas iteraciones y pruebas exhaustivas sobre el lenguaje.

El tipado es un tema recurrente en la academia, los beneficios y desventajas de los lenguajes estáticos y dinámicos son estudiados y conocidos en profundidad. Además, existen muchos enfoques que buscan combinar los beneficios de ambos mundos, gradual typing es uno de ellos.

Los sistemas de tipos son la base fundamental de los chequeadores de tipos, además, son la documentación por excelencia.

5.1. Trabajo a futuro

El proyecto fue sufriendo ajustes que llevaron a resolver ciertos problemas y posponer otros.

En primer lugar, para estar completamente seguros de nuestra formalización, todavía tenemos que demostrar propiedades de nuestro sistema de tipos, como la robustez o preservación de tipos. Para eso, primero necesitamos formalizar la semántica operativa del lenguaje. En un sistema como el que presentamos el progreso no se mantiene porque algunas verificaciones (originadas por *downcasts*) se delegan al tiempo de ejecución, y es posible que fallen produciendo que la reducción se atasque. Sin embargo, creemos que el progreso se mantendría en nuestro sistema para los programas bien redactados y totalmente verificados estáticamente (es decir, programas que no requieren *downcasts*). También, se debería formalizar la meta teoría del sistema de tipos.

Por otro lado, en nuestro contexto, definir una función pseudopolimórfica implica perder las propiedades de tipado de la función. Como trabajo futuro, planeamos introducir el polimorfismo paramétrico adecuado a nuestro sistema de tipos.

Como se ha demostrado a lo lago del informe, el sistema no es extensivo a todo el lenguaje por lo que otra línea de desarrollo futuro es ampliar las características que estamos considerando. En particular, se destacan algunos operadores muy utilizados en el desarrollo diario como lo son los operadores pipe ($|>$), pin ($\wedge$), *with*, *when*, *use* o *require*, entre otros. Además, soportar valores por defecto en las firmas de las funciones, tener un conjunto de funciones en el preludio de Elixir para las cuales ya se conozcan sus tipos o permitir tipos de datos definidos por el usuario mediante la notación *@type*.

Por último, si el sistema de tipos se introduce en el flujo de compilación del lenguaje, se podría investigar como utilizar dicha información para realizar optimizaciones en el código que se genera.

Glosario

algoritmo Conjunto ordenado de operaciones sistemáticas que permite hacer un cálculo y hallar la solución de un tipo de problemas. 4, 7, 10

AST (Árbol de sintaxis abstracta) Representación en árbol de la estructura sintáctica abstracta del código fuente escrito en un lenguaje de programación. Cada nodo del árbol denota una construcción que ocurre en el código fuente. 5, 52, 53

bottom type Es el tipo que no tiene valores. También se le llama tipo cero o vacío. 14

bottom-up ('de abajo arriba') Estrategia de procesamiento de información característica de las ciencias de la información. Las partes individuales se diseñan con detalle y luego se enlazan para formar componentes más grandes, que a su vez se enlazan hasta que se forma el sistema completo. 14

build Construcción de algo (usualmente una aplicación) que tiene un resultado observable y tangible. 6

camel case Se refiere al estilo de escritura en sin espacios ni puntuación. Lo que indica la separación de palabras es una sola letra en mayúscula. 21

channels Modelo para la comunicación entre procesos y la sincronización mediante el paso de mensajes. 5

concurrencia Habilidad de distintas partes de un programa, algoritmo, o problema de ser ejecutado en desorden o en orden parcial, sin afectar el resultado final. 1, 3

contravariante Invierte el orden de la covarianza. 24, 49

covariante Conserva el orden de los tipos, y los ordena de más específicos a más genéricos. 24, 25, 49

downcast Reducir una referencia de una clase base a una de sus clases derivadas. 26, 51, 59

EBNF (Backus–Naur Form) Notación formal para definir la sintaxis de un lenguaje. Usada para especificar la mayoría de los lenguajes de programación. 14

framework Abstracción en la que el software que proporciona una funcionalidad genérica se puede cambiar de forma selectiva mediante código adicional escrito por el usuario, proporcionando así software específico de la aplicación. 5

inferencia Se refiere a la detección automática del tipo de datos de una expresión en un lenguaje de programación. 4, 9, 10, 12, 56

lexer Herramienta para llevar a cabo el análisis léxico, el cual es el proceso de convertir una secuencia de caracteres en una secuencia de tokens. 5, 52

literal Notación para representar un valor fijo. 14, 15, 19, 20, 34, 37, 38, 41, 43, 45

metaprogramación Técnica de programación en la que los programas de computadora tienen la capacidad de tratar otros programas como sus datos. Significa que un programa puede diseñarse para leer, generar, analizar o transformar otros programas, e incluso modificarse a sí mismo mientras se ejecuta. 5

nil Significa "nada" o la ausencia de algún objeto. 15

open source Programas informáticos que permiten el acceso a su código de programación, lo que facilita modificaciones por parte de otros programadores ajenos a los creadores originales del software en cuestión. 53

paradigma Principios fundamentales de la programación de software. Se los puede ver como estilos de programación fundamentalmente diferenciados que, en consecuencia, generan códigos de software que están estructurados de forma distinta. 1, 3, 7

parser Herramienta para llevar a cabo el análisis sintáctico, el cual es el proceso de analizar una cadena de símbolos, ya sea en lenguaje natural, lenguajes de computadora o estructuras de datos, conforme a las reglas de una gramática formal. 5, 52

pattern-matching Es un mecanismo para comparar un valor con un patrón. Un emparejamiento exitoso entre patrón y valor permite deconstruir el valor en las partes que lo constituyen. 5, 17

performance Medida o cuantificación de la velocidad/resultado con que se realiza una tarea o proceso. 9

polling Operación de consulta constante, generalmente hacia un dispositivo de hardware, para crear una actividad sincrónica sin el uso de interrupciones, aunque también puede suceder lo mismo para recursos de software. 5

pruebas funcionales Prueba de tipo caja negra basada en la ejecución, revisión y retroalimentación de las funcionalidades previamente diseñadas para el software. 56

pruebas unitarias Es una forma de comprobar el correcto funcionamiento de una unidad de código. 8, 56

schedulers Es un módulo del sistema operativo que selecciona los siguientes trabajos que se admitirán en el sistema y el siguiente proceso que se ejecutará. 4

script Programa usualmente simple, que por lo general se almacena en un archivo de texto plano. 21, 28

snake case Se refiere al estilo de escritura en el que cada espacio se reemplaza por un guión bajo (_) y la primera letra de cada palabra es escrita en minúscula. 15

top type Es el tipo que contiene todos los objetos posibles en el sistema de tipos de interés. 12, 14, 46, 58

web sockets Tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP. Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse por cualquier aplicación cliente/servidor. 5

Apéndice A

Reglas

$$\frac{}{t <: t}\ \text{(ST\_REFL)} \qquad \frac{t_1 <: t_2 \quad t_2 <: t_3}{t_1 <: t_3}\ \text{(ST\_TRANS)}$$

$$\frac{}{\texttt{none} <: t}\ \text{(ST\_NONE)} \qquad \frac{}{t <: \texttt{any}}\ \text{(ST\_ANY)}$$

$$\frac{}{\texttt{integer} <: \texttt{float}}\ \text{(ST\_NUM)} \qquad \frac{t <: u}{\texttt{[}t\texttt{]} <: \texttt{[}u\texttt{]}}\ \text{(ST\_LIST)}$$

$$\frac{t_1 <: u_1 \quad \cdots \quad t_n <: u_n}{\texttt{\{}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{\}} <: \texttt{\{}u_1\texttt{,}\ldots\texttt{,}u_n\texttt{\}}}\ \text{(ST\_TUPLE)}$$

$$\frac{k' <: k \qquad u_1 <: u'_1 \quad \cdots \quad u_n <: u'_n}{\texttt{\%\{}k \Rightarrow u_1\texttt{,}\ldots\texttt{,}k \Rightarrow u_{n+m}\texttt{\}} <: \texttt{\%\{}k' \Rightarrow u'_1\texttt{,}\ldots\texttt{,}k' \Rightarrow u'_n\texttt{\}}}\ \text{(ST\_MAP)}$$

$$\frac{u_1 <: t_1 \quad \cdots \quad u_n <: t_n \qquad t <: u}{(t_1\texttt{,}\ldots\texttt{,}t_n) \to t <: (u_1\texttt{,}\ldots\texttt{,}u_n) \to u}\ \text{(ST\_FUN)}$$

Figura A.1: Reglas de subtipado

$$\frac{}{t <| t}\ \text{(LS\_REFL)} \qquad \frac{t_1 <| t_2 \quad t_2 <| t_3}{t_1 <| t_3}\ \text{(LS\_TRANS)}$$

$$\frac{}{\texttt{none} <| t}\ \text{(LS\_NONE)} \qquad \frac{}{\texttt{integer} <| \texttt{float}}\ \text{(LS\_NUM)}$$

$$\frac{t <| u}{[t] <| [u]}\ \text{(LS\_LIST)}$$

$$\frac{t_1 <| u_1 \quad \cdots \quad t_n <| u_n}{\{t_1, \ldots, t_n\} <| \{u_1, \ldots, u_n\}}\ \text{(LS\_TUPLE)}$$

$$\frac{k' <: k \qquad u_1 <| u'_1 \quad \cdots \quad u_n <| u'_n}{\%\{k \Rightarrow u_1, \ldots, k \Rightarrow u_{n+m}\} <| \%\{k' \Rightarrow u'_1, \ldots, k' \Rightarrow u'_n\}}\ \text{(LS\_MAP)}$$

$$\frac{u_1 <| t_1 \quad \cdots \quad u_n <| t_n \qquad t <| u}{(t_1, \ldots, t_n) \to t <| (u_1, \ldots, u_n) \to u}\ \text{(LS\_FUN)}$$

Figura A.2: Reglas de subtipado limitado

$$\frac{}{t \lll t}\ \text{(PR\_REFL)} \qquad \frac{}{t \lll any}\ \text{(PR\_ANY)}$$

$$\frac{t \lll u}{[t] \lll [u]}\ \text{(PR\_LIST)}$$

$$\frac{t_1 \lll u_1 \quad \cdots \quad t_n \lll u_n}{\{t_1, \ldots, t_n\} \lll \{u_1, \ldots, u_n\}}\ \text{(PR\_TUPLE)}$$

$$\frac{k \lll k' \qquad u_1 \lll u'_1 \quad \cdots \quad u_n \lll u'_n}{\%\{k \Rightarrow u_1, \ldots, k \Rightarrow u_n\} \lll \%\{k' \Rightarrow u'_1, \ldots, k' \Rightarrow u'_n\}}\ \text{(PR\_MAP)}$$

$$\frac{t_1 \lll u_1 \quad \cdots \quad t_n \lll u_n \qquad t \lll u}{(t_1, \ldots, t_n) \to t \lll (u_1, \ldots, u_n) \to u}\ \text{(PR\_FUN)}$$

Figura A.3: Reglas de precisión

$$\frac{\varnothing; \epsilon \vdash^r m \Rightarrow \Delta \quad \Delta; \epsilon \vdash^{ch} m}{\vdash^p m}\ \text{(P\_M)} \qquad \frac{\varnothing; \varnothing; \epsilon \vdash^t e : t \Rightarrow \Gamma}{\vdash^p e}\ \text{(P\_E)}$$

Figura A.4: Reglas de programa correctamente tipado

$$\frac{\Delta;\rho \vdash^r m_1 \Rightarrow \Delta^1 \quad \Delta^1;\rho \vdash^r m_2 \Rightarrow \Delta^2}{\Delta;\rho \vdash^r m_1; m_2 \Rightarrow \Delta^2}\ \text{(R\_MS)}$$

$$\frac{\Delta;\rho.m\_name \vdash^r m \Rightarrow \Delta^1}{\Delta;\rho \vdash^r \texttt{defmodule}\ m\_name\ \texttt{do}\ m\ \texttt{end} \Rightarrow \Delta^1}\ \text{(R\_M)}$$

Figura A.5: Reglas de recorrida de módulos

$$\frac{\Delta;\rho \vdash^r f_1 \Rightarrow \Delta^1 \quad \Delta^1;\rho \vdash^r f_2 \Rightarrow \Delta^2}{\Delta;\rho \vdash^r f_1; f_2 \Rightarrow \Delta^2}\ \text{(R\_FS)}$$

$$\frac{(\rho.f\_name, n) \notin \Delta}{\begin{array}{l}\Delta;\rho \vdash^r \texttt{@spec}\ f\_name(t_1, ..., t_n)\ \texttt{::}\ t \\ \qquad \Rightarrow \Delta[(\rho.f\_name, n) \mapsto \texttt{(}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{)} \to t]\end{array}}\ \text{(R\_FSPEC)}$$

$$\frac{}{\Delta;\rho \vdash^r \texttt{def[p]}\ f\_name(p_1, ..., p_n)\ \texttt{do}\ e\ \texttt{end} \Rightarrow \Delta}\ \text{(R\_FDEF)}$$

Figura A.6: Reglas de recorrida de funciones

$$\frac{}{\Delta;\rho \vdash^r e \Rightarrow \Delta}\ \text{(R\_E)}$$

Figura A.7: Reglas de recorrida de expresiones

$$\frac{\Delta;\rho \vdash^{ch} m_1 \quad \Delta;\rho \vdash^{ch} m_2}{\Delta;\rho \vdash^{ch} m_1; m_2}\ \text{(CH\_MS)}$$

$$\frac{\Delta;\rho.m\_name \vdash^{ch} m}{\Delta;\rho \vdash^{ch} \texttt{defmodule}\ m\_name\ \texttt{do}\ m\ \texttt{end}}\ \text{(CH\_MM)}$$

$$\frac{\Delta;\rho.m\_name \vdash^{ch} f}{\Delta;\rho \vdash^{ch} \texttt{defmodule}\ m\_name\ \texttt{do}\ f\ \texttt{end}}\ \text{(CH\_MF)}$$

$$\frac{\Delta;\varnothing;\rho.m\_name \vdash^t e : t \Rightarrow \Gamma}{\Delta;\rho \vdash^{ch} \texttt{defmodule}\ m\_name\ \texttt{do}\ e\ \texttt{end}}\ \text{(CH\_ME)}$$

Figura A.8: Reglas de chequeo de módulos

$$\frac{\Delta; \rho \vdash^{ch} f_1 \quad \Delta; \rho \vdash^{ch} f_2}{\Delta; \rho \vdash^{ch} f_1; f_2} \text{ (CH\_FS)}$$

$$\frac{\begin{array}{c}(\rho.f\_name, n) \in \Delta \\ \Delta(\rho.f\_name, n) = (t_1, \ldots, t_n) \to t \\ \varnothing \vdash^{tp} p_1 : t'_1 \Rightarrow \Gamma^1 \\ \cdots \\ \Gamma^{n-1} \vdash^{tp} p_n : t'_n \Rightarrow \Gamma^n \\ \Delta; \Gamma^n; \rho \vdash^{t} e : t \Rightarrow \Gamma' \\ t'_1 <: t_1 \ \ldots \ t'_n <: t_n\end{array}}{\Delta; \rho \vdash^{ch} \texttt{def[p]}\ f\_name(p_1, ..., p_n)\ \texttt{do}\ e\ \texttt{end}} \text{ (CH\_FN)}$$

$$\frac{(\rho.f\_name, n) \notin \Delta}{\Delta; \rho \vdash^{ch} \texttt{def[p]}\ f\_name(p_1, ..., p_n)\ \texttt{do}\ e\ \texttt{end}} \text{ (CH\_FN\_NT)}$$

Figura A.9: Reglas de chequeo de funciones

$$\frac{\Gamma \vdash^{tp} p : t \Rightarrow \Gamma^1 \quad t <| u}{\Gamma \vdash^{tp} p : u \Rightarrow \Gamma^1} \text{ (TP\_SUB)}$$

Figura A.10: Reglas de tipado de subsumption para patrones

$$\frac{}{\Gamma \vdash^{tp} f : \texttt{float} \Rightarrow \Gamma} \text{ (TP\_F)} \qquad \frac{}{\Gamma \vdash^{tp} a : \texttt{atom} \Rightarrow \Gamma} \text{ (TP\_A)}$$

$$\frac{}{\Gamma \vdash^{tp} s : \texttt{string} \Rightarrow \Gamma} \text{ (TP\_S)} \qquad \frac{}{\Gamma \vdash^{tp} b : \texttt{boolean} \Rightarrow \Gamma} \text{ (TP\_B)}$$

$$\frac{}{\Gamma \vdash^{tp} i : \texttt{integer} \Rightarrow \Gamma} \text{ (TP\_I)}$$

Figura A.11: Reglas de tipado de patrones literales

$$\frac{x \in \Gamma \quad \Gamma(x) = t}{\Gamma \vdash^{tp} x : t \Rightarrow \Gamma} \text{ (TP\_VAR)} \qquad \frac{x \notin \Gamma}{\Gamma \vdash^{tp} x : t \Rightarrow \Gamma[x \mapsto t]} \text{ (TP\_VARN)}$$

Figura A.12: Reglas de tipado de patrones variables

$$\frac{}{\Gamma \vdash^{tp} \texttt{[ ]} : \texttt{[}t\texttt{]} \Rightarrow \Gamma}\ (\text{TP\_ELIST})$$

$$\frac{\begin{array}{c}\Gamma \vdash^{tp} p_1 : t \Rightarrow \Gamma^1 \\ \Gamma^1 \vdash^{tp} p_2 : \texttt{[}t\texttt{]} \Rightarrow \Gamma^2\end{array}}{\Gamma \vdash^{tp} \texttt{[}p_1 \texttt{|} p_2\texttt{]} : \texttt{[}t\texttt{]} \Rightarrow \Gamma^2}\ (\text{TP\_LIST})$$

Figura A.13: Reglas de tipado de patrones listas

$$\frac{}{\Gamma \vdash^{tp} \texttt{\%\{ \}} : \texttt{\%\{ \}} \Rightarrow \Gamma}\ (\text{TP\_EMAP})$$

$$\frac{\begin{array}{c}\Gamma \vdash^{ktp} kp_1 : t \\ \cdots \\ \Gamma \vdash^{ktp} kp_n : t \\ \Gamma \vdash^{tp} p_1 : u_1 \Rightarrow \Gamma^1 \\ \cdots \\ \Gamma^{n-1} \vdash^{tp} p_n : u_n \Rightarrow \Gamma^n\end{array}}{\Gamma \vdash^{tp} \texttt{\%\{}kp_1 \Rightarrow p_1 \texttt{,} \ldots \texttt{,} kp_n \Rightarrow p_n\texttt{\}} : \texttt{\%\{}t \Rightarrow u_1 \texttt{,} \ldots \texttt{,} t \Rightarrow u_n\texttt{\}} \Rightarrow \Gamma^n}\ (\text{TP\_MAP})$$

Figura A.14: Reglas de tipado de patrones mapa

$$\frac{\begin{array}{c}\Gamma \vdash^{tp} p_1 : t_1 \Rightarrow \Gamma^1 \\ \cdots \\ \Gamma^{n-1} \vdash^{tp} p_n : t_n \Rightarrow \Gamma^n\end{array}}{\Gamma \vdash^{tp} \texttt{\{}p_1 \texttt{,} \ldots \texttt{,} p_n\texttt{\}} : \texttt{\{}t_1 \texttt{,} \ldots \texttt{,} t_n\texttt{\}} \Rightarrow \Gamma^n}\ (\text{TP\_TUP})$$

Figura A.15: Reglas de tipado de patrones tuplas

$$\frac{}{\Gamma \vdash^{tp} \texttt{\_} : t \Rightarrow \Gamma}\ (\text{TP\_WILD})$$

Figura A.16: Reglas de tipado de patrones wildcard

$$\frac{\begin{array}{c}\Gamma \vdash^{tp} p_1 : t \Rightarrow \Gamma^1 \\ \Gamma^1 \vdash^{tp} p_2 : t \Rightarrow \Gamma^2\end{array}}{\Gamma \vdash^{tp} p_1 = p_2 : t \Rightarrow \Gamma^2}\ (\text{TP\_BIND})$$

Figura A.17: Reglas de tipado de patrones bind

$$\frac{\Gamma \vdash^{ktp} p : t \qquad t <| u}{\Gamma \vdash^{ktp} p : u}\ \text{(KTP\_SUB)}$$

Figura A.18: Reglas de tipado de subsumption para claves de patrones mapas

$$\frac{}{\Gamma \vdash^{ktp} f : \texttt{float}}\ \text{(KTP\_F)} \qquad \frac{}{\Gamma \vdash^{ktp} a : \texttt{atom}}\ \text{(KTP\_A)}$$

$$\frac{}{\Gamma \vdash^{ktp} s : \texttt{string}}\ \text{(KTP\_S)} \qquad \frac{}{\Gamma \vdash^{ktp} b : \texttt{boolean}}\ \text{(KTP\_B)}$$

$$\frac{}{\Gamma \vdash^{ktp} i : \texttt{integer}}\ \text{(KTP\_I)}$$

Figura A.19: Reglas de tipado de literales en claves de patrones mapas

$$\frac{}{\Gamma \vdash^{ktp} \texttt{[ ]} : \texttt{[}t\texttt{]}}\ \text{(KTP\_ELIST)}$$

$$\frac{\Gamma \vdash^{ktp} p_1 : t \qquad \Gamma \vdash^{ktp} p_2 : \texttt{[}t\texttt{]}}{\Gamma \vdash^{ktp} \texttt{[}p_1 \texttt{|} p_2\texttt{]} : \texttt{[}t\texttt{]}}\ \text{(KTP\_LIST)}$$

Figura A.20: Reglas de tipado de listas en claves de patrones mapas

$$\frac{}{\Gamma \vdash^{ktp} \texttt{\%\{ \}} : \texttt{\%\{ \}}}\ \text{(KTP\_EMAP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{ktp} kp_1 : t \\ \cdots \\ \Gamma \vdash^{ktp} kp_n : t \\ \Gamma \vdash^{kltp} p_1 : u_1 \\ \cdots \\ \Gamma \vdash^{ktp} p_n : u_n\end{array}}{\Gamma \vdash^{ktp} \texttt{\%\{}kp_1 \Rightarrow p_1\texttt{,} \ldots \texttt{,}kp_n \Rightarrow p_n\texttt{\}} : \texttt{\%\{}t \Rightarrow u_1\texttt{,} \ldots \texttt{,}t \Rightarrow u_n\texttt{\}}}\ \text{(KTP\_MAP)}$$

Figura A.21: Reglas de tipado de mapas en claves de patrones mapas

$$\frac{\begin{array}{c}\Gamma \vdash^{ktp} p_1 : t_1 \\ \cdots \\ \Gamma \vdash^{ktp} p_n : t_n\end{array}}{\Gamma \vdash^{ktp} \{p_1, \ldots, p_n\} : \{t_1, \ldots, t_n\}} \text{(KTP\_TUP)}$$

Figura A.22: Reglas de tipado de tuplas en claves de patrones mapas

$$\frac{\begin{array}{c}\Gamma \vdash^{ktp} p_1 : t \\ \Gamma \vdash^{ktp} p_2 : t\end{array}}{\Gamma \vdash^{ktp} p_1 = p_2 : t} \text{(KTP\_BIND)}$$

Figura A.23: Reglas de tipado de binds en claves de patrones mapas

$$\frac{\Delta; \Gamma; \rho \vdash^{t} e : t \Rightarrow \Gamma^1 \quad t <| \, u}{\Delta; \Gamma; \rho \vdash^{t} e : u \Rightarrow \Gamma^1} \text{(TE\_SUB)}$$

$$\frac{\Delta; \Gamma; \rho \vdash^{t} e : t \Rightarrow \Gamma^1 \quad u \lll t}{\Delta; \Gamma; \rho \vdash^{t} e : u \Rightarrow \Gamma^1} \text{(TE\_DOWN)}$$

Figura A.24: Reglas de tipado de expresiones subsumption y downcast

$$\frac{}{\Delta; \Gamma; \rho \vdash^{t} f : \texttt{float} \Rightarrow \Gamma} \text{(TE\_F)} \qquad \frac{}{\Delta; \Gamma; \rho \vdash^{t} a : \texttt{atom} \Rightarrow \Gamma} \text{(TE\_A)}$$

$$\frac{}{\Delta; \Gamma; \rho \vdash^{t} s : \texttt{string} \Rightarrow \Gamma} \text{(TE\_S)} \qquad \frac{}{\Delta; \Gamma; \rho \vdash^{t} b : \texttt{boolean} \Rightarrow \Gamma} \text{(TE\_B)}$$

$$\frac{}{\Delta; \Gamma; \rho \vdash^{t} i : \texttt{integer} \Rightarrow \Gamma} \text{(TE\_I)}$$

Figura A.25: Reglas de tipado de expresiones literales

$$\frac{\begin{array}{c}x \in \Gamma \\ \Gamma(x) = t\end{array}}{\Delta; \Gamma; \rho \vdash^{t} x : t \Rightarrow \Gamma} \text{(TE\_VAR)}$$

Figura A.26: Reglas de tipado de expresiones variables

$$\frac{}{\Delta;\Gamma;\rho \vdash^t \texttt{[ ]} : \texttt{[any]} \Rightarrow \Gamma}\ (\text{TE\_ELIST})$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : t \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : \texttt{[}t\texttt{]} \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t \texttt{[}e_1\texttt{|}e_2\texttt{]} : \texttt{[}t\texttt{]} \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ (\text{TE\_LIST})$$

Figura A.27: Reglas de tipado de expresiones listas

$$\frac{}{\Delta;\Gamma;\rho \vdash^t \texttt{\%\{ \}} : \texttt{\%\{ \}} \Rightarrow \Gamma}\ (\text{TE\_EMAP})$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : t \Rightarrow \Gamma^1 \\ \cdots \\ \Delta;\Gamma;\rho \vdash^t e_n : t \Rightarrow \Gamma^n \\ \Delta;\Gamma;\rho \vdash^t e'_1 : u_1 \Rightarrow \Gamma'^1 \\ \cdots \\ \Delta;\Gamma;\rho \vdash^t e'_n : u_n \Rightarrow \Gamma'^n\end{array}}{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t \texttt{\%\{ }e_1 \Rightarrow e'_1\texttt{,}\ldots\texttt{,}e_n \Rightarrow e'_n\texttt{\}} : \texttt{\%\{}t \Rightarrow u_1\texttt{,}\ldots\texttt{,}t \Rightarrow u_n\texttt{\}} \\ \Rightarrow \Gamma^1 \cup_R \Gamma'^1 \cup_R \ldots \cup_R \Gamma^n \cup_R \Gamma'^n\end{array}}\ (\text{TE\_MAP})$$

Figura A.28: Reglas de tipado de expresiones mapa

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \cdots \\ \Delta;\Gamma;\rho \vdash^t e_n : t_n \Rightarrow \Gamma^n\end{array}}{\Delta;\Gamma;\rho \vdash^t \texttt{\{}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{\}} : \texttt{\{}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{\}} \Rightarrow \Gamma^1 \cup_R \ldots \cup_R \Gamma^n}\ (\text{TE\_TUP})$$

Figura A.29: Reglas de tipado de expresiones tuplas

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e : t \Rightarrow \Gamma^1 \\ \varnothing \vdash^{tp} p : t \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t p = e : t \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ (\text{TE\_BIND})$$

Figura A.30: Reglas de tipado de expresiones bind

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \Delta;\Gamma^1;\rho \vdash^t e_2 : t_2 \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1; e_2 : t_2 \Rightarrow \Gamma^2}\ (\text{TE\_ES})$$

Figura A.31: Reglas de secuencia de expresiones

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : t \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : t \Rightarrow \Gamma^2 \\ t <: \texttt{float}\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1 \diamond e_2 : t \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ \text{(TE\_ARITH)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : \texttt{float} \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : \texttt{float} \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1 / e_2 : \texttt{float} \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ \text{(TE\_DIV)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e : t \Rightarrow \Gamma^1 \\ t <: \texttt{float}\end{array}}{\Delta;\Gamma;\rho \vdash^t \texttt{-}e : t \Rightarrow \Gamma^1}\ \text{(TE\_NEG)}$$

Figura A.32: Reglas de expresiones aritméticas

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : \texttt{boolean} \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : \texttt{boolean} \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1 \bullet e_2 : \texttt{boolean} \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ \text{(TE\_BOP)}$$

$$\frac{\Delta;\Gamma;\rho \vdash^t e : \texttt{boolean} \Rightarrow \Gamma^1}{\Delta;\Gamma;\rho \vdash^t \texttt{not}\ e : \texttt{boolean} \Rightarrow \Gamma^1}\ \text{(TE\_NOT)}$$

Figura A.33: Reglas de expresiones booleanas

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : t_2 \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1 \star e_2 : \texttt{boolean} \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ \text{(TE\_CMP)}$$

Figura A.34: Reglas de expresiones de operaciones de comparacion

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : \texttt{[}t\texttt{]} \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : \texttt{[}t\texttt{]} \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1 \square e_2 : \texttt{[}t\texttt{]} \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ \text{(TE\_LOP)}$$

Figura A.35: Reglas de expresiones concatenación de listas

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : \texttt{\%\{}t \Rightarrow u_1\texttt{,...,}t \Rightarrow u_n\texttt{\}} \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : t \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1\texttt{[}e_2\texttt{]} : \texttt{any} \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ \text{(TE\_MAPAPP)}$$

Figura A.36: Reglas de expresiones de aplicación de mapas

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : \texttt{string} \Rightarrow \Gamma^1 \\ \Delta;\Gamma;\rho \vdash^t e_2 : \texttt{string} \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t e_1 \texttt{<>} e_2 : \texttt{string} \Rightarrow \Gamma^1 \cup_R \Gamma^2}\ \text{(TE\_CONCAT)}$$

Figura A.37: Reglas de expresiones de operaciones con cadenas de caracteres

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e : \texttt{boolean} \Rightarrow \Gamma' \\ \Delta;\Gamma';\rho \vdash^t e_1 : t \Rightarrow \Gamma^1\end{array}}{\Delta;\Gamma;\rho \vdash^t \odot\ e\ \texttt{do}\ e_1\ \texttt{end} : t \Rightarrow \Gamma'}\ \text{(TE\_IF/U)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e : \texttt{boolean} \Rightarrow \Gamma' \\ \Delta;\Gamma';\rho \vdash^t e_1 : t \Rightarrow \Gamma^1 \\ \Delta;\Gamma';\rho \vdash^t e_2 : t \Rightarrow \Gamma^2\end{array}}{\Delta;\Gamma;\rho \vdash^t \odot\ e\ \texttt{do}\ e_1\ \texttt{else}\ e_2\ \texttt{end} : t \Rightarrow \Gamma'}\ \text{(TE\_IF/U\_ELSE)}$$

Figura A.38: Reglas de expresiones if/unless

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e : t \Rightarrow \Gamma' \\ \varnothing \vdash^{tp} p_1 : t \Rightarrow \Gamma^1 \\ \cdots \\ \varnothing \vdash^{tp} p_n : t \Rightarrow \Gamma^2 \\ \Delta;\Gamma' \cup_R \Gamma^1;\rho \vdash^t e_1 : t' \Rightarrow \Gamma'^1 \\ \cdots \\ \Delta;\Gamma' \cup_R \Gamma^n;\rho \vdash^t e_n : t' \Rightarrow \Gamma'^n\end{array}}{\Delta;\Gamma;\rho \vdash^t \texttt{case}\ e\ \texttt{do}\ p_1 \to e_1, ..., p_n \to e_n\ \texttt{end} : t' \Rightarrow \Gamma'}\ \text{(TE\_CASE)}$$

Figura A.39: Reglas de expresiones case

$$\frac{\begin{array}{c}\Delta;\Gamma;\rho \vdash^t e_1 : \texttt{boolean} \Rightarrow \Gamma^1 \\ \cdots \\ \Delta;\Gamma;\rho \vdash^t e_n : \texttt{boolean} \Rightarrow \Gamma^n \\ \Delta;\Gamma^1;\rho \vdash^t e'_1 : t \Rightarrow \Gamma^{1'} \\ \cdots \\ \Delta;\Gamma^n;\rho \vdash^t e'_n : t \Rightarrow \Gamma^{n'}\end{array}}{\Delta;\Gamma;\rho \vdash^t \texttt{cond do}\ e_1 \to e'_1, ..., e_n \to e'_n\ \texttt{end} : t \Rightarrow \Gamma}\ \text{(TE\_COND)}$$

Figura A.40: Reglas de expresiones cond

$$\frac{\begin{array}{c}\Delta(\rho_2.f\_name, n) = (t_1, \ldots, t_n) \to t \\ \Delta; \Gamma; \rho_1 \vdash^t e_1 : t'_1 \Rightarrow \Gamma^1 \\ \ldots \\ \Delta; \Gamma; \rho_1 \vdash^t e_n : t'_n \Rightarrow \Gamma^n \\ t'_1 <: t_1 \quad \ldots \quad t'_n <: t_n\end{array}}{\Delta; \Gamma; \rho_1 \vdash^t \rho_2.f\_name\ (e_1, ..., e_n) : t \Rightarrow \Gamma^1 \cup_R ... \cup_R \Gamma^n}\ \text{(TE\_APPFE)}$$

$$\frac{\begin{array}{c}(\rho_2.f\_name, n) \notin \Delta \\ \Delta; \Gamma; \rho_1 \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \ldots \\ \Delta; \Gamma; \rho_1 \vdash^t e_n : t_n \Rightarrow \Gamma^n\end{array}}{\Delta; \Gamma\rho_1 \vdash^t \rho_2.f\_name\ (e_1, ..., e_n) : \texttt{any} \Rightarrow \Gamma^1 \cup_R ... \cup_R \Gamma^n}\ \text{(TE\_NAPPFE)}$$

$$\frac{\begin{array}{c}\Delta(\rho.f\_name, n) = (t_1, \ldots, t_n) \to t \\ \Delta; \Gamma; \rho \vdash^t e_1 : t'_1 \Rightarrow \Gamma^1 \\ \ldots \\ \Delta; \Gamma; \rho \vdash^t e_n : t'_n \Rightarrow \Gamma^n \\ t'_1 <: t_1 \quad \ldots \quad t'_n <: t_n\end{array}}{\Delta; \Gamma; \rho \vdash^t f\_name\ (e_1, ..., e_n) : t \Rightarrow \Gamma^1 \cup_R ... \cup_R \Gamma^n}\ \text{(TE\_APPFI)}$$

$$\frac{\begin{array}{c}(\rho.f\_name, n) \notin \Delta \\ \Delta; \Gamma; \rho \vdash^t e_1 : t_1 \Rightarrow \Gamma^1 \\ \ldots \\ \Delta; \Gamma; \rho \vdash^t e_n : t_n \Rightarrow \Gamma^n\end{array}}{\Delta; \Gamma\rho \vdash^t f\_name\ (e_1, ..., e_n) : \texttt{any} \Rightarrow \Gamma^1 \cup_R ... \cup_R \Gamma^n}\ \text{(TE\_NAPPFI)}$$

Figura A.41: Reglas de expresiones de llamadas a funciones

Apéndice B

Proceso de compilación

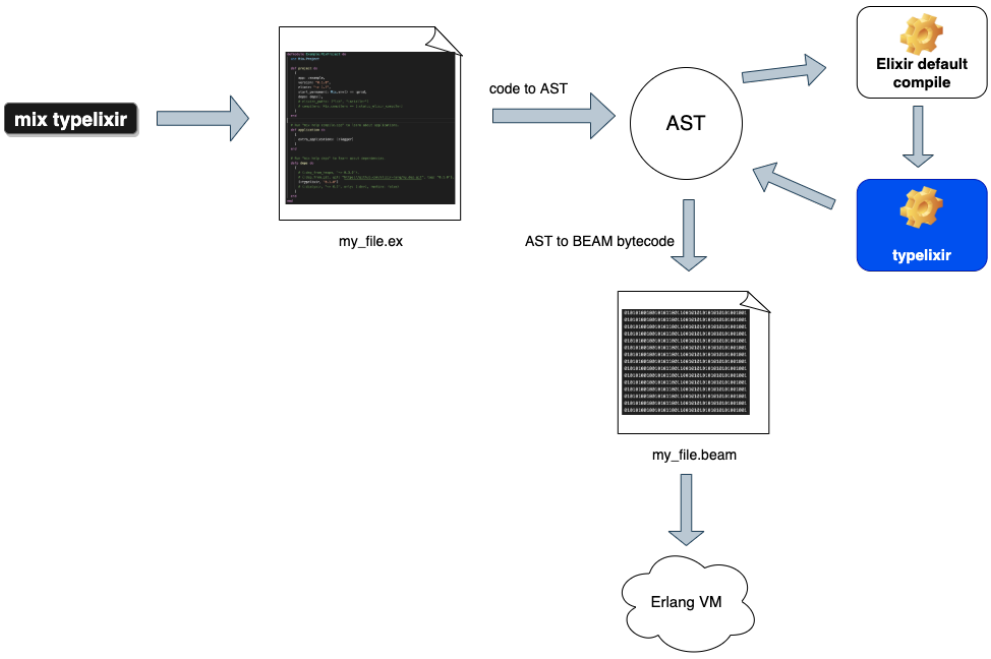


Figura B.1: Proceso de compilación

Apéndice C

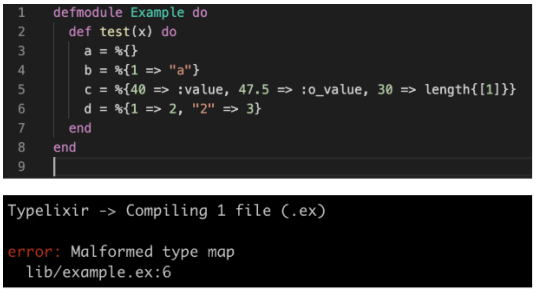
Ejemplos de uso de la biblioteca

```
1   defmodule Example do
2     def test(x) do
3       a = [1, 2]
4       b = [1, length([])]
5       c = [1, 40.0]
6       d = []
7       e = [1, 2, :a]
8     end
9   end
10
```

```
Typelixir -> Compiling 1 file (.ex)

error: Malformed type list
  lib/example.ex:
```

Figura C.1: Mal formación de listas

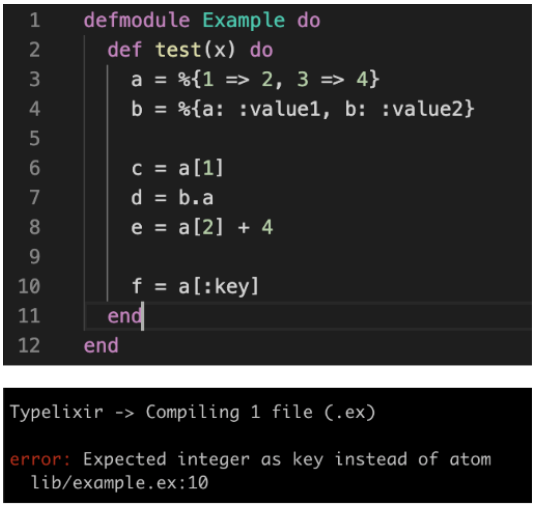


```
defmodule Example do
  def test(x) do
    a = %{}
    b = %{1 => "a"}
    c = %{40 => :value, 47.5 => :o_value, 30 => length{[1]}}
    d = %{1 => 2, "2" => 3}
  end
end
```

```
Typelixir -> Compiling 1 file (.ex)

error: Malformed type map
  lib/example.ex:6
```

Figura C.2: Mal formación de mapas



```
defmodule Example do
  def test(x) do
    a = %{1 => 2, 3 => 4}
    b = %{a: :value1, b: :value2}

    c = a[1]
    d = b.a
    e = a[2] + 4

    f = a[:key]
  end
end
```

```
Typelixir -> Compiling 1 file (.ex)

error: Expected integer as key instead of atom
  lib/example.ex:10
```

Figura C.3: Aplicación de mapas

```
1   defmodule Example do
2     def test() do
3       a = 1 + 2
4       b = 2 + 3.4
5       c = 1 / 2
6       d = length([]) + 2
7       e = 4 + "5"
8     end
9   end
10
```

```
Typelixir -> Compiling 1 file (.ex)

error: Type error on + operator
  lib/example.ex:7
```

Figura C.4: Operadores aritméticos

```
1   defmodule Example do
2     def test() do
3       if (true), do: 10
4       if (true), do: 10, else: 10
5
6       if (false) do
7         true
8       else
9         10
10      end
11    end
12  end
13
```

```
Typelixir -> Compiling 1 file (.ex)

error: Type error on if branches
  lib/example.ex:6
```

Figura C.5: Ramas del if

```elixir
defmodule Example do
  @spec test(string) :: integer
  def test(x) do
    10
  end

  def test(x) do
    length([]) # any
  end

  def test(x) do
    "not pass"
  end
end
```

```
Typelixir -> Compiling 1 file (.ex)

error: Body doesn't match function type on test/1 declaration
  lib/example.ex:11
```

Figura C.6: Cuerpos de función

```elixir
defmodule Example do
  @spec test(integer, string) :: integer
  def test(x, y), do: x
  def test([x], y), do: 10
  def test(x, x), do: 11

  @spec test2({integer, boolean}) :: integer
  def test({x, y}), do: x
  def test2([1, 2]), do: 10
  def test2({1, true, 3}), do: 11
end

```

```
Typelixir -> Compiling 1 file (.ex)

error: Parameters does not match type specification
  lib/example.ex:4

error: Variable x is already defined with type integer
  lib/example.ex:5

error: Parameters does not match type specification
  lib/example.ex:9

error: The number of parameters in tuple does not match the number of types
  lib/example.ex:10
```

Figura C.7: Parámetros de función

```elixir
defmodule Example do
  defmodule Example2 do
    @spec test(integer, string) :: string
    def test(x, y), do: y
  end

  @spec test(integer) :: string
  def test(1), do: UnknownModule.test(1)
  def test(2), do: Example.Example2.test(2, "a")
  def test(4), do: Example.Example2.test(true, "a")
  def test(x), do: Example.Example2.test(x, {1, 2})

  @spec test2(integer, integer) :: string
  def test2(x, y), do: Example.test(1)
  def test2(1, 2), do: Example.test(true)
end
```

```
Typelixir -> Compiling 1 file (.ex)

error: Arguments does not match type specification on test/2
  lib/example.ex:10

error: Arguments does not match type specification on test/2
  lib/example.ex:11

error: Arguments does not match type specification on test/1
  lib/example.ex:15
```

Figura C.8: Llamados de función

Apéndice D

Respuesta entrevista José Valim

"*Hi Mauricio and Agustin,*

Thanks for reaching out to me, this looks fantastic! Congratulations on getting the paper approved too!

Me and the Elixir team have always considered static typing throughout the years. We even developed a prototype for a static language with intersection types, which turned out to be impractical really fast, and from them on we decided to explore gradual typing. And that's what we have been doing over the last year or so.

However, instead of jumping head first into a typed language, we have decided to first add some basic type inference to the language. Our current mindset is to implement a mini-dialyzer: one that is fast, has good error messages, and always runs. Our goal with these changes is to explore the impact typing will have on the language and the community without changing any of Elixir's public API. This effort is not only about typing either, we want to generally improve the general compile-time guarantees of an Elixir codebase.

If you look at the CHANGELOG for the next Elixir release, you can see some of this new behaviour described there: `https://github.com/elixir-lang/elixir/blob/master/CHANGELOG.md`

In other words, we are very interested in this direction and your work and inputs will definitely be very appreciated. In particular, Eric (from the Elixir team) is currently the one working on typing and both of us lack the theoretical background that would validate our typing/inference rules are correct. And while we don't want to add a type signature API right now (i.e. we don't want to use @spec nor add our own), this is something that is definitely on the line in the long term. We also want to explore typing of structs, behaviours, and protocols, as well as visibility of struct and modules.

To answer your questions more directly, my understanding is that we are moving towards the same direction. We decided to focus on some basic inference

for now but our end goals are similar. I believe this would indeed be useful for large Elixir codebases and we foresee it even being part of the language, rather than a separate library. I am not quite sure yet on what would be my recommendation for future work but I will be glad to further discuss it.

That said, given this background, how do you think is the best way to move forward? :) If I may ask, what are your plans for the project? Is this something you are going to directly work on over the next year so (i.e. as part of a master thesis or similar)?

Thanks again and have a great sunday!"

Apéndice E

Derivaciones sobre el sistema de tipos

$$\frac{}{\Delta;\Gamma;\rho \vdash^{t} 2.3 : \texttt{float} \Rightarrow \Gamma}\ (\text{TE\_F}) \qquad \frac{}{\Delta;\Gamma;\rho \vdash^{t} \text{:foo} : \texttt{atom} \Rightarrow \Gamma}\ (\text{TE\_A})$$

$$\frac{}{\Delta;\Gamma;\rho \vdash^{t} \text{"}foo\text{"}: \texttt{string} \Rightarrow \Gamma}\ (\text{TE\_S}) \qquad \frac{}{\Delta;\Gamma;\rho \vdash^{t} \text{true} : \texttt{boolean} \Rightarrow \Gamma}\ (\text{TE\_B})$$

$$\frac{}{\Delta;\Gamma;\rho \vdash^{t} 23 : \texttt{integer} \Rightarrow \Gamma}\ (\text{TE\_I})$$

Figura E.1: Ejemplos reglas de tipado de expresiones literales

$$\frac{\begin{array}{c}\mathrm{x} \in \Gamma \\ \Gamma(\ \mathrm{x}\ ) = \texttt{integer}\end{array}}{\Delta; \Gamma = \{\ \mathrm{x} \Rightarrow \texttt{integer}\}; \rho \vdash^{t} \mathrm{x} : \texttt{integer} \Rightarrow \Gamma}\ (\text{TE\_VAR})$$

Figura E.2: Ejemplo regla de tipado de expresiones variables

$$\frac{\dfrac{}{\Delta; \Gamma; \rho \vdash^{t} \mathrm{x} : \texttt{integer} \Rightarrow \Gamma}\ (\text{TE\_VAR}) \qquad \dfrac{}{\Delta; \Gamma; \rho \vdash^{t} \texttt{[ ]} : \texttt{[integer]} \Rightarrow \Gamma}\ (\text{TE\_ELIST})}{\Delta; \Gamma = \{\ \mathrm{x} \Rightarrow \texttt{integer}\}; \rho \vdash^{t} \texttt{[} x \texttt{]} : \texttt{[integer]} \Rightarrow \Gamma}\ (\text{TE\_LIST})$$

$$\frac{\dfrac{}{\Delta; \Gamma; \rho \vdash^{t} 23 : \texttt{integer} \Rightarrow \Gamma}\ (\text{TE\_I}) \qquad \Delta; \Gamma; \rho \vdash^{t} \texttt{[} x \texttt{]} : \texttt{[integer]} \Rightarrow \Gamma}{\Delta; \Gamma = \{\ \mathrm{x} \Rightarrow \texttt{integer}\}; \rho \vdash^{t} \texttt{[23 | [} x \texttt{]]} : \texttt{[integer]} \Rightarrow \Gamma}\ (\text{TE\_LIST})$$

Figura E.3: Ejemplo reglas de tipado de expresiones listas

$$\frac{\begin{array}{c}\Delta; \Gamma; \rho \vdash^{t} \text{:foo} : \texttt{atom} \Rightarrow \Gamma \\ \Delta; \Gamma; \rho \vdash^{t} \text{:boo} : \texttt{atom} \Rightarrow \Gamma \\ \Delta; \Gamma; \rho \vdash^{t} 23 : \texttt{integer} \Rightarrow \Gamma \\ \Delta; \Gamma; \rho \vdash^{t} \mathrm{x} : \texttt{integer} \Rightarrow \Gamma\end{array}}{\Delta; \Gamma = \{\ \mathrm{x} \Rightarrow \texttt{integer}\}; \rho \vdash^{t} \texttt{\%\{ :}foo \Rightarrow 23\texttt{, :}boo \Rightarrow x\texttt{\}} : \texttt{\%\{ atom} \Rightarrow \texttt{integer, atom} \Rightarrow \texttt{integer \}} \Rightarrow \Gamma}\ (\text{TE\_MAP})$$

Figura E.4: Ejemplo reglas de tipado de expresiones mapa

$$\frac{\dfrac{}{\Delta; \Gamma; \rho \vdash^{t} 23 : \texttt{integer} \Rightarrow \Gamma}\ (\text{TE\_I}) \qquad \dfrac{}{\Delta; \Gamma; \rho \vdash^{t} \text{:foo} : \texttt{atom} \Rightarrow \Gamma}\ (\text{TE\_A})}{\Delta; \Gamma = \{\ \mathrm{x} \Rightarrow \texttt{integer}\}; \rho \vdash^{t} \texttt{\{ 23, :}foo\texttt{\}} : \texttt{\{ integer, atom\}} \Rightarrow \Gamma}\ (\text{TE\_TUP})$$

Figura E.5: Ejemplo reglas de tipado de expresiones tuplas

$$\frac{\dfrac{x \notin \varnothing}{\varnothing \vdash^{tp} \mathrm{x} : \texttt{integer} \Rightarrow \{\ \mathrm{x} \Rightarrow \texttt{integer}\}}\ (\text{TP\_VARN}) \qquad \dfrac{y \notin \varnothing}{\varnothing \vdash^{tp} \mathrm{z} : \texttt{atom} \Rightarrow \{\ \mathrm{z} \Rightarrow \texttt{atom}\}}\ (\text{TP\_VARN})}{\varnothing \vdash^{tp} \texttt{\{}x\texttt{, } z\texttt{\}} : \texttt{\{ integer, atom\}} \Rightarrow \{\ \mathrm{x} \Rightarrow \texttt{integer}, \mathrm{z} \Rightarrow \texttt{atom}\}}\ (\text{TP\_TUP})$$

Figura E.6: Ejemplo reglas de tipado de patrones tuplas

$$\frac{\dfrac{}{\Delta; \Gamma; \rho \vdash^{t} \text{:foo} : \texttt{atom} \Rightarrow \Gamma}\ (\text{TE\_A}) \qquad \dfrac{y \notin \varnothing}{\varnothing \vdash^{tp} \mathrm{y} : \texttt{atom} \Rightarrow \{\ \mathrm{y} \Rightarrow \texttt{atom}\}}\ (\text{TP\_VARN})}{\Delta; \Gamma = \{\ \mathrm{x} \Rightarrow \texttt{integer}\}; \rho \vdash^{t} \mathrm{y} = \text{:foo} : \texttt{atom} \Rightarrow \{\ \mathrm{x} \Rightarrow \texttt{integer}, \mathrm{y} \Rightarrow \texttt{atom}\}}\ (\text{TE\_BIND})$$

$$\frac{\dfrac{}{\Delta; \Gamma; \rho \vdash^{t} \texttt{\{ 23, :}boo\texttt{\}} : \texttt{\{ integer, atom\}} \Rightarrow \Gamma}\ (\text{TE\_A}) \qquad \dfrac{}{\varnothing \vdash^{tp} \texttt{\{}x\texttt{, } z\texttt{\}} : \texttt{\{ integer, atom\}} \Rightarrow \{\ \mathrm{x} \Rightarrow \texttt{integer}, \mathrm{z} \Rightarrow \texttt{atom}\}}\ (\text{TP\_TUP})}{\Delta; \Gamma = \{\ \mathrm{x} \Rightarrow \texttt{integer}\}; \rho \vdash^{t} \texttt{\{}x\texttt{, } z\texttt{\}} = \texttt{\{ 23, :}boo\texttt{\}} : \texttt{\{ integer, atom\}} \Rightarrow \{\ \mathrm{x} \Rightarrow \texttt{integer}, \mathrm{z} \Rightarrow \texttt{atom}\}}\ (\text{TE\_BIND})$$

Figura E.7: Ejemplo reglas de tipado de expresiones bind

$$\dfrac{\dfrac{}{\Delta;\varnothing;\rho \vdash^t \text{x} = 23 : \texttt{integer} \Rightarrow \{x \Rightarrow \texttt{integer}\}}\,(\text{TE\_BIND}) \quad \dfrac{}{\Delta;\Gamma = \{\text{x} \Rightarrow \texttt{integer}\};\rho \vdash^t \text{z} = \text{:foo} : \texttt{atom} \Rightarrow \{\text{x} \Rightarrow \texttt{integer}, \text{z} \Rightarrow \texttt{atom}\}}\,(\text{TE\_BIND})}{\Delta;\varnothing;\rho \vdash^t \text{x} = 23; \text{z} = \text{:foo} : \texttt{atom} \Rightarrow \{x \Rightarrow \texttt{integer}, z \Rightarrow \texttt{atom}\}}\,(\text{TE\_ES})$$

Figura E.8: Ejemplo reglas de secuencia de expresiones

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t 23 : \texttt{integer} \Rightarrow \Gamma}\,(\text{TE\_I}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{x} : \texttt{integer} \Rightarrow \Gamma}\,(\text{TE\_VAR}) \quad \texttt{integer} <: \texttt{float}}{\Delta;\Gamma = \{x \Rightarrow \texttt{integer}\};\rho \vdash^t 23 + \text{x} : \texttt{integer} \Rightarrow \Gamma}\,(\text{TE\_ARITH})$$

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t 23 : \texttt{float} \Rightarrow \Gamma}\,(\text{TE\_SUB}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{x} : \texttt{float} \Rightarrow \Gamma}\,(\text{TE\_SUB})}{\Delta;\Gamma = \{x \Rightarrow \texttt{integer}\};\rho \vdash^t 23 \;/\; \text{x} : \texttt{float} \Rightarrow \Gamma}\,(\text{TE\_DIV})$$

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{x} : \texttt{integer} \Rightarrow \Gamma}\,(\text{TE\_VAR}) \quad \texttt{integer} <: \texttt{float}}{\Delta;\Gamma = \{x \Rightarrow \texttt{integer}\};\rho \vdash^t \text{-x} : \texttt{integer} \Rightarrow \Gamma}\,(\text{TE\_NEG})$$

Figura E.9: Ejemplo reglas de expresiones aritméticas

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{true} : \texttt{boolean} \Rightarrow \Gamma}\,(\text{TE\_B}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{y} : \texttt{boolean} \Rightarrow \Gamma}\,(\text{TE\_VAR})}{\Delta;\Gamma = \{y \Rightarrow \texttt{boolean}\};\rho \vdash^t \text{true and x} : \texttt{boolean} \Rightarrow \Gamma}\,(\text{TE\_BOP})$$

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{y} : \texttt{boolean} \Rightarrow \Gamma}\,(\text{TE\_VAR})}{\Delta;\Gamma = \{y \Rightarrow \texttt{boolean}\};\rho \vdash^t \texttt{not y} : \texttt{boolean} \Rightarrow \Gamma}\,(\text{TE\_NOT})$$

Figura E.10: Ejemplo reglas de expresiones booleanas

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{:foo} : \texttt{atom} \Rightarrow \Gamma}\,(\text{TE\_A}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{x} : \texttt{integer} \Rightarrow \Gamma}\,(\text{TE\_VAR})}{\Delta;\Gamma = \{x \Rightarrow \texttt{integer}\};\rho \vdash^t \text{x == :foo} : \texttt{boolean} \Rightarrow \Gamma}\,(\text{TE\_CMP})$$

Figura E.11: Ejemplo de reglas de expresiones de operaciones de comparacion

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \texttt{[} x \texttt{]} : \texttt{[integer]} \Rightarrow \Gamma}\,(\text{TE\_VAR}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{z} : \texttt{[integer]} \Rightarrow \Gamma}\,(\text{TE\_VAR})}{\Delta;\Gamma = \{z \Rightarrow \texttt{[integer]}\};\rho \vdash^t \text{z ++ [} x \text{]} : \texttt{[integer]} \Rightarrow \Gamma}\,(\text{TE\_LOP})$$

Figura E.12: Ejemplo reglas de expresiones concatenación de listas

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{m} : \texttt{\%\{ atom} \Rightarrow \texttt{integer, atom} \Rightarrow \texttt{integer \}} \Rightarrow \Gamma}\,(\text{TE\_VAR}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{:foo} : \texttt{atom} \Rightarrow \Gamma}\,(\text{TE\_A})}{\Delta;\Gamma = \{m \Rightarrow \texttt{\%\{ atom} \Rightarrow \texttt{integer, atom} \Rightarrow \texttt{integer \}}\};\rho \vdash^t \text{m}[:foo] : \texttt{any} \Rightarrow \Gamma}\,(\text{TE\_MAPAPP})$$

Figura E.13: Ejemplo reglas de expresiones de aplicación de mapas

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{k} : \texttt{string} \Rightarrow \Gamma}\ (\text{TE\_VAR}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{"}foo\text{"} : \texttt{string} \Rightarrow \Gamma}\ (\text{TE\_S})}{\Delta;\Gamma = \{k \Rightarrow \texttt{string}\};\rho \vdash^t \text{k <> "}foo\text{"} : \texttt{string} \Rightarrow \Gamma}\ (\text{TE\_CONCAT})$$

Figura E.14: Ejemplo reglas de expresiones de operaciones con cadenas de caracteres

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{b} : \texttt{boolean} \Rightarrow \Gamma}\ (\text{TE\_B}) \quad \dfrac{}{\Delta;\Gamma';\rho \vdash^t \text{:boo} : \texttt{atom} \Rightarrow \Gamma}\ (\text{TE\_A}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{x} = 23;\ \text{:foo} : \texttt{atom} \Rightarrow \{b \Rightarrow \texttt{boolean}, x \Rightarrow \texttt{integer}\}}\ (\text{TE\_ES})}{\Delta;\Gamma = \{b \Rightarrow \texttt{boolean}\};\rho \vdash^t \texttt{if b do x} = 23;\ \text{:foo } \texttt{else :boo end} : \texttt{atom} \Rightarrow \Gamma}\ (\text{TE\_IF/U\_ELSE})$$

Figura E.15: Ejemplo reglas de expresiones if/unless

$$\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{a} : \texttt{atom} \Rightarrow \Gamma}\ (\text{TE\_VAR}) \quad \dfrac{\dfrac{}{\varnothing \vdash^{tp} \text{:foo} : \texttt{atom} \Rightarrow \Gamma}\ (\text{TP\_A})}{\varnothing \vdash^{tp} \text{:boo} : \texttt{atom} \Rightarrow \Gamma}\ (\text{TP\_A}) \quad \dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{x} = 23;\ \text{false} : \texttt{boolean} \Rightarrow \{a \Rightarrow \texttt{atom}, x \Rightarrow \texttt{integer}\}}\ (\text{TE\_ES})}{\Delta;\Gamma;\rho \vdash^t \text{true} : \texttt{boolean} \Rightarrow \Gamma}\ (\text{TE\_B})}{\Delta;\Gamma = \{a \Rightarrow \texttt{atom}\};\rho \vdash^t \texttt{case a do :foo} \rightarrow \text{true, :boo} \rightarrow \text{x} = 23;\ \text{false } \texttt{end} : \texttt{boolean} \Rightarrow \Gamma}\ (\text{TE\_CASE})$$

Figura E.16: Ejemplo reglas de expresiones case

$$\dfrac{\dfrac{\dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{b} : \texttt{boolean} \Rightarrow \Gamma}\ (\text{TE\_VAR}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t \text{y} : \texttt{boolean} \Rightarrow \Gamma}\ (\text{TE\_VAR})}{\Delta;\Gamma;\rho \vdash^t \text{true} : \texttt{boolean} \Rightarrow \Gamma}\ (\text{TE\_B}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t 1 : \texttt{integer}}\ (\text{TE\_I}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t 2 : \texttt{integer}}\ (\text{TE\_I}) \quad \dfrac{}{\Delta;\Gamma;\rho \vdash^t 3 : \texttt{integer}}\ (\text{TE\_I})}{\Delta;\Gamma = \{y \Rightarrow \texttt{boolean}, b \Rightarrow \texttt{boolean}\};\rho \vdash^t \texttt{cond do y} \rightarrow 1\ ,\ \text{b} \rightarrow 2\ ,\ \text{true} \rightarrow 3\ \texttt{end} : \texttt{integer} \Rightarrow \Gamma}\ (\text{TE\_COND})$$

Figura E.17: Ejemplo reglas de expresiones cond

$$\dfrac{\Delta(B.foo, 1) = \texttt{(string)} \rightarrow \texttt{atom} \quad \texttt{string} <: \texttt{string} \qquad \dfrac{}{\Delta; \Gamma; A \vdash^t \texttt{"}boo\texttt{"}: \texttt{string} \Rightarrow \Gamma}\ (\text{TE\_S})}{\Delta = \{(B.foo, 1) \Rightarrow \texttt{(string)} \rightarrow \texttt{atom}\}; \Gamma; A \vdash^t \text{B.foo}(\texttt{"}boo\texttt{"}): \texttt{atom} \Rightarrow \Gamma}\ (\text{TE\_APPFE})$$

$$\dfrac{(C.foo, 1) \notin \Delta \qquad \dfrac{}{\Delta; \Gamma; A \vdash^t \texttt{"}boo\texttt{"}: \texttt{string} \Rightarrow \Gamma}\ (\text{TE\_S})}{\Delta = \{(B.foo, 1) \Rightarrow \texttt{(string)} \rightarrow \texttt{atom}\}; \Gamma; A \vdash^t \text{C.foo}(\texttt{"}boo\texttt{"}) : \texttt{any} \Rightarrow \Gamma}\ (\text{TE\_NAPPFE})$$

$$\dfrac{\Delta(B.foo, 1) = \texttt{(string)} \rightarrow \texttt{atom} \quad \texttt{string} <: \texttt{string} \qquad \dfrac{}{\Delta; \Gamma; B \vdash^t \texttt{"}boo\texttt{"}: \texttt{string} \Rightarrow \Gamma}\ (\text{TE\_S})}{\Delta = \{(B.foo, 1) \Rightarrow \texttt{(string)} \rightarrow \texttt{atom}\}; \Gamma; B \vdash^t \text{foo}(\texttt{"}boo\texttt{"}): \texttt{atom} \Rightarrow \Gamma}\ (\text{TE\_APPFI})$$

$$\dfrac{(A.foo, 1) \notin \Delta \qquad \dfrac{}{\Delta; \Gamma; A \vdash^t \texttt{"}boo\texttt{"}: \texttt{string} \Rightarrow \Gamma}\ (\text{TE\_S})}{\Delta = \{(B.foo, 1) \Rightarrow \texttt{(string)} \rightarrow \texttt{atom}\}; \Gamma; A \vdash^t \text{foo}(\texttt{"}boo\texttt{"}) : \texttt{any} \Rightarrow \Gamma}\ (\text{TE\_NAPPFI})$$

Figura E.18: Ejemplo reglas de expresiones de llamadas a funciones

Bibliografía

Cardelli, L. (2004). Type systems. En A. B. Tucker (Ed.), *Computer science handbook* (cap. 97). Chapman & Hall/CRC.

Cartwright, R., y Fagan, M. (2004). Soft typing. *SIGPLAN Not.*, *39*(4), 412–428.

Cassola, M., Talagorria, A., Pardo, A., y Viera, M. (2020). A gradual type system for elixir. En *Proceedings of the 24th brazilian symposium on context-oriented programming and advanced modularity* (p. 17–24). New York, NY, USA: Association for Computing Machinery. Recuperado de `https://doi.org/10.1145/3427081.3427084`

Castagna, G., y Lanvin, V. (2017, agosto). Gradual typing with union and intersection types. *Proc. ACM Program. Lang.*, *1*(ICFP). Recuperado de `https://doi.org/10.1145/3110285` doi: 10.1145/3110285

Castagna, G., Lanvin, V., Petrucciani, T., y Siek, J. G. (2019). Gradual typing: A new perspective. *Proc. ACM Program. Lang.*, *3*(POPL).

Claessen, K., y Hughes, J. (2011). Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, *46*(4), 53-64.

Dialyxir. (2020). *Dialyxir repository.* https://github.com/jeremyjh/dialyxir.

Gradualizer. (2020). *Gradualizer.* https://github.com/josefs/Gradualizer.

Hex. (2020). *Hex.* https://hex.pm/.

Hudak, P., Hughes, J., Peyton Jones, S., y Wadler, P. (2007). A history of haskell: Being lazy with class. En *Proceedings of the third acm sigplan conference on history of programming languages* (p. 12–1–12–55). New York, NY, USA: Association for Computing Machinery. Recuperado de `https://doi.org/10.1145/1238844.1238856` doi: 10.1145/1238844.1238856

Jensen, S. H., Møller, A., y Thiemann, P. (2009). Type analysis for javascript. En *Proceedings of the 16th international symposium on static analysis* (p. 238–255). Berlin, Heidelberg: Springer-Verlag. Recuperado de `https://doi.org/10.1007/978-3-642-03237-0_17` doi: 10.1007/978-3-642-03237-0_17

Jurić, S. (2015). *Elixir in action.* Shelter Island, NY: Manning Publications.

Lindahl, T., y Sagonas, K. (2006). Practical type inference based on success typings. En *Proceedings of the 8th acm sigplan international conference on principles and practice of declarative programming* (p. 167–178). New

York, NY, USA: ACM Press.
McCord, C. (2015). *Metaprogramming elixir: Write less code, get more done (and have fun!).* Raleigh, North Carolina: The Pragmatic Bookshelf.
McCord, C. (2019). *Programming phoenix: Productive - reliable - fast.* Raleigh, North Carolina: The Pragmatic Bookshelf.
Meijer, E., y Drayton, P. (2004). Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages..
Mix. (2020). *Mix.* https://hexdocs.pm/mix/Mix.html.
Rémy, D. (2013). *Type systems for programming languages.* http://gallium.inria.fr/ remy/mpri/2013/cours.pdf.
Sagonas, K., y Luna, D. (2008). Gradual typing of Erlang programs: A wrangler experience. En *Proceedings of the 7th ACM SIGPLAN workshop on Erlang* (p. 73-82). New York, NY, USA: ACM Press.
Siek, J., y Taha, W. (2006). Gradual typing for functional languages. En *Scheme and functional programming workshop.*
Siek, J., y Taha, W. (2007). Gradual typing for objects. En E. Ernst (Ed.), *ECOOP 2007 - Object-Oriented Programming* (p. 2-27). Berlin, Heidelberg: Springer Berlin Heidelberg.
Stenman, E. (2018). Beam: A virtual machine for handling millions of messages per second (invited talk). En *Proceedings of the 10th acm sigplan international workshop on virtual machines and intermediate languages* (p. 4). New York, NY, USA: ACM Press.
Thatte, S. (1989). Quasi-static typing. En *Proceedings of the 17th acm sigplan-sigact symposium on principles of programming languages* (p. 367–381). New York, NY, USA: Association for Computing Machinery.
Thomas, D. (2014). *Programming elixir: Functional - concurrent - pragmatic - fun* (1.ª ed.). Pragmatic Bookshelf.
Thomas, D., Fowler, C., y Hunt, A. (2013). *Programming ruby 1.9 & 2.0: The pragmatic programmers' guide* (4.ª ed.). Dallas, TX: Pragmatic Bookshelf.
Typelixir. (2020). *Typelixir.* https://github.com/Typelixir/typelixir.