UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE
INGENIERÍA

Un sistema de tipos gradual con semántica de casts para el lenguaje Elixir

Informe de Proyecto de Grado presentado por

Damián Ferencz

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Alberto Pardo
Marcos Viera

Montevideo, 10 de junio de 2023

Agradecimientos

A mis padres y mis hermanos, por ayudarme, de una manera u otra, a llegar a este momento. También a Alicia y a mis abuelos. Realmente para mí no hubiera sido posible sin el apoyo y el aguante; yo se que fueron muchos años recibiendo la misma respuesta: «Me recibo el año que viene».

A los tutores Marcos y Alberto por la oportunidad y por el trabajo. Fueron dos años intensos para ellos también; demasiadas horas en reuniones de zoom, correos y hasta algún que otro whatsapp. Me han acompañado durante la etapa más difícil de la carrera, pero también la más transformadora.

Finalmente, una mención especial al barbilla color sal y pimienta. Fiel y exigente compañero; de los incontables paseos obligados de la tarde y de la noche han surgido ideas claves para que esto tomara forma.

Resumen

En este proyecto se desarrolla un sistema de tipos para un fragmento importante de Elixir, un lenguaje de programación funcional de uso industrial con tipado dinámico que se ejecuta sobre la máquina virtual de Erlang. El sistema propuesto tiene la característica de ser algorítmico y estar basado en subtipado estructural. Además, incorpora a su diseño la filosofía del *gradual typing*, con lo que se hace posible restringir los chequeos de tipo a las porciones anotadas del programa.

En el trabajo también se introduce una nueva semántica de evaluación para los programas chequeados con la que se consigue garantizar que el comportamiento de las porciones anotadas del código se preserva durante la ejecución sin cambios con respecto al sistema estático. La semántica se basa en una etapa de inserción de casts de tipo en posiciones críticas del código fuente, que ocurre *a posteriori* del chequeo de tipos y previo a la compilación.

Se presenta también una implementación concreta del sistema de tipos. El proceso de inserción de casts es implementado como una traducción al mismo lenguaje ahora enriquecido con una nueva directiva para casts de tipo, que se consigue representar dentro del lenguaje utilizando las herramientas nativas que Elixir provee para la metaprogramación.

Palabras clave: Elixir, *gradual typing*, type systems

Índice general

Capítulo 1

Introducción

Elixir (Juric [2019]) es un lenguaje de programación funcional de propósito general que se ejecuta en la máquina virtual de Erlang (BEAM). Fue creado en 2012 por José Valim con un foco puesto en la productividad del desarrollador, brindar soporte para la concurrencia y la programación web ágil. 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 et al. [2013]). Elixir está orientado hacia el desarrollo de aplicaciones industriales, por lo que cuenta con un conjunto de herramientas que simplifica la gestión de proyectos, las pruebas, el empaquetado y la creación de documentación.

Al estar basado en BEAM, el código de Elixir corre dentro de procesos livianos que pueden ser ejecutados concurrentemente en de una misma máquina o paralelamente dentro de un *cluster*, lo que permite a sus aplicaciones ser fácilmente escalables. Siendo un lenguaje funcional, se promueve un estilo de programación que produce código conciso y mantenible, además de que simplifica el desarrollo de programas concurrentes gracias a la inmutabilidad forzosa en las asignaciones.

Elixir, al igual que Erlang, es un lenguaje de tipado dinámico. Existen herramientas, como *Dialyxir*, que permiten cierto grado de chequeo estático. Dialyxir funciona conectándose con una herramienta de Erlang llamada Dialyzer, que realiza un análisis estático sobre el *bytecode* generado para la BEAM. Dialyzer se basa en el concepto de *success typing* de Lindahl and Sagonas [2006] para encontrar errores de tipos en el código y como tal, tiene la desventaja de ser bastante permisivo, mientras que es flexible y más adaptable al estilo de escritura dinámica. Como el código BEAM no se mapea al código Elixir original, el *output* del análisis no permite identificar fácilmente la ubicación de los errores ni sus causas, y esto hace que no resulte tan útil para los desarrolladores.

Una manera muy difundida para mitigar los errores de programación en lenguajes compilados es mediante el uso de sistemas de tipos estáticos (Pierce [2002]). Estos permiten ejecutar chequeos sobre el codigo fuente durante la etapa de compilación, con lo que se consiguen detecciones tempranas de errores de programación. En general, los sistemas de tipos estáticos no son suficientes para garantizar el uso correcto de las abstracciones de un lenguaje y resultan necesarios chequeos por parte del *runtime system* para prevenir algunas fallas durante la ejecución -por ejemplo, para evitar el acceso a indices incorrectos dentro de un arreglo-. No obstante, los chequeos de tipos permiten capturar gran parte de los escenarios comunes de error y focalizar los esfuerzos en prevenir aquellos que les escapan.

Además de la prevención de errores, la adopción de sistemas de tipos estáticos suele tener efectos muy favorables para el ecosistema del lenguaje en cuestión. Por un lado, incentivan el uso de patrones de abstracción y composición comunes que ayudan a la mantenibilidad de las bases de código de gran escala. Por otro

lado, constituyen una base fundamental para el diseño de herramientas que facilitan la actividad de los desarrolladores, como búsquedas masivas, refactorizaciones complejas y generación de documentación automática. Finalmente, el chequeo de tipos genera información que suele ser aprovechada por el propio compilador para optimizar el código de máquina generado.

En vista de los beneficios del tipado estático (i.e., en compilación), en los últimos años varios de lenguajes de tipado dinámico han propuesto sistemas de tipos (*Typescript*, *Dart*, *Erlang*, *Clojure*), aunque en general éstos han sido diseñados para poder convivir con código existente y, por lo tanto, permiten ignorar los chequeos en ciertas porciones del código. En general, este enfoque permite combinar las ventajas del tipado dinámico como pueden ser la flexibilidad y facilidad para el prototipado, con todos los beneficios discutidos del tipado en compilación, además de que posibilita la adopción gradual del tipado en bases de código de gran tamaño.

En base a la popularidad en aumento de las variantes de tipado "opcional", han surgido algunas lineas de investigación dedicadas a formular y refinar los principios sobre los que éstas se manifiestan. Una de las que ha tenido mayor impacto es la del *gradual typing* (Siek and Taha [2006]), que toma como punto de partida la teoría existente para los sistema de tipos éstaticos e incoprora dos nuevos conceptos clave: el *tipo dinámico* y la *relación de consistencia* entre tipos. El tipo dinámico permite "tipar" las porciones no chequeadas del código, mientras que la relación de consistencia permite definir una noción de compatibilidad entre tipos que contempla la precencia del tipo dinámico. Añadiendo el tipo dinámico a la sintaxis e integrando la relación de consistencia en las reglas del sistema, se hacen posibles los chequeos estáticos sobre codigo que se compone de partes estáticas y dinámicas.

Más que un mecanismo para extender los chequeos a un contexto de tipos opcionales, la disciplina del *gradual typing* busca extender la experiencia completa del tipado estático hacia cualquier programa parcialmente tipado. En partícular, en un sistema gradual los tipos estáticos deben proveer las mismas garantías en el *runtime*, y esto obliga a redefinir la semántica del lenguaje con el fin de evitar fallas dinámicas dentro de las pociones estáticas de un programa. Dentro de esa nueva semántica, las fallas dinámicas pueden ocurrir tanto en las porciones dinámicas como en los puntos de transición hacia porciones estáticas; de ésta forma, toda anotación de código se asegura de ser respetada durante la ejecución.

En este trabajo se desarrolla un sistema de tipos gradual para Elixir. Toma como punto de partida el sistema gradual introducido en Cassola et al. [2022] para un fragmento considerable del lenguaje que incluye soporte para los constructos sintácticos básicos -en particular, una primitiva de *pattern matching* y la definición de casos basada en patrones-, además de mecanismos para el alto orden. En ese trabajo, el foco estaba puesto en los aspectos estáticos del tipado gradual y su contribución fue un sistema de tipos basado en especificaciones de tipos para las funciones y una versión gradual de ese sistema que contempla el escenario de funciones no especificadas. Aquí, la gradualización del sistema anterior se completa con la formalización de una nueva semántica de evaluación para el fragmento del lenguaje con base en la definición de un proceso de inserción de casts de tipo que se nutre de la información producida por el chequeo exitoso para un programa dado. Adicionalmente, se presenta una implementación concreta para el sistema de tipos definido.

El resto del informe se organiza en 4 captiulos. En el Capítulo 2 se especifica el fragmento de Elixir considerado para el trabajo - que es ligeramente distinto al de Cassola et al. [2022]- y se describe informalmente la semántica de evaluación con la ayuda de ejemplos. En el Capítulo 3 se reformula el sistema de tipos definido en Cassola et al. [2022] en base a una presentación algorítmica; esto supone cambios importantes tanto en la definición de las reglas del sistema estático como en el mecanismo de extensión gradual, que en este caso toma ideas de Cimini and Siek [2016] y Garcia et al. [2016]. El sistema resultante se evalúa a la luz de la *Static Gradual Guarantee* de Siek et al. [2015], concretamente, adaptando el enunciando para el contexto actual y presentando un bosquejo de la prueba formal. El sistema de tipos algorítmico es el punto de partida para la definición, en el Capítulo 4, de un proceso de traducción similar al de Siek and Taha [2007] que opera composicionalmente sobre la sintaxis de los programas chequeados, insertando casts de tipo alrededor de algunas

expresiones críticas. La definición de la nueva semántica se construye a partir de la definición de un modelo de evaluación para el fragmento de Elixir que contempla los casts de tipo insertados por la traducción. En el Capítulo 5 se hace una breve dscripción de la implementación concreta de los sistemas de chequeo y traducción de los capítulos anteriores, y en particular de los mecanismos de metaprogramación en Elixir que hicieron posible la representación de los casts dentro del mismo lenguaje. Finalmente, en el Capítulo 6 se hacen algunas reflexiones finales y se consideran propuestas -con variado grado de detalle- que podrían ser interesantes para tomar en cuenta en un trabajo futuro.

Capítulo 2

Introduccion al lenguaje Elixir

El presente capítulo tiene por objetivo introducir el fragmento del lenguaje que se utilizará como base para este trabajo. Se busca formalizar la jerarquía sintáctica que constituye dicho fragmento a la vez que familiarizar al lector con sus peculiaridades sintácticas y de evaluación.

2.1. Generalidades

El fragmento de la sintaxis escogido es suficientemente rico como para ofrecer una muestra del interés que puede resultar del estudio de Elixir como lenguaje de programación funcional. Como tal, se decidió incluir declaraciones de función, estructuras de datos, estructuras de control y alguna de las directivas que hacen posible el alto orden.

No se hará énfasis en las características del lenguaje que posibilitan la programación modular y la aplicación de patrones de encapsulamiento. Es posible incluir módulos y funciones privadas e integrarlas sin gran dificultad a lo desarrollado en los capítulos siguientes; esto forma parte de las contribuciones de Cassola et al. [2022]. Por otra parte, los *protocols* permiten implementar polimorfismo paramétrico al nivel de los tipos de datos y resultan de interés, por lo que se consideran interesantes para un trabajo futuro.

Tampoco se hará mención a las facilidades del lenguaje que permiten dar soporte a la programación en múltiples hilos. En esa línea, se omitirán las directivas que permiten definir y manipular procesos de la máquina virtual BEAM dentro de Elixir. Este aspecto es el foco del trabajo de Tabone and Francalanza [2021].

Finalmente, se dejan de lado las definiciones de guardas, de *structs* -para declarar registros- y declaraciones de funciones anónimas, que, a pesar de pertenecer a la "porción funcional" y ser relevantes, suponen retos técnicos y de alcance que obligan a ser dejados fuera de éste trabajo.

2.1.1. Sintaxis

Un ejecutable en Elixir puede estar definido en uno o más módulos; que a su vez pueden estar distribuidos dentro de uno o varios archivos. Los módulos se componen típicamente de una lista de declaraciones, donde cada una define una función. En general, se distingue un módulo principal que contiene una declaración `main` que funciona como punto de entrada al ejecutable. Cada declaración consta de una firma con cero o más parámetros de entrada y un cuerpo que puede depender de las variables que ocurren en los parámetros. En la mayoría de los casos, dicho cuerpo toma la forma particular de una secuencia de expresiones separadas por saltos de linea.

Las expresiones en Elixir, como es habitual en otros lenguajes, incluyen valores literales, variables, asignaciones y llamadas a funciones. A partir de estas expresiones básicas se pueden construir expresiones más complejas como tuplas y listas, pero también estructuras de control, como los condicionales, y definición por casos.

Las asignaciones en Elixir son especiales en cuanto admiten el uso de *patrones* -y no únicamente variables- del lado izquierdo. Esto se conoce en la jerga del lenguaje como *pattern matching*. Los patrones asumen una forma particular (como tupla o lista) para la expresión del lado derecho y permiten ligar variables dentro de subexpresiones anidadas.

Elixir es un lenguaje de alto orden, i.e. , permite representar funciones al nivel de las expresiones. Para dicho propósito incluye en su sintaxis un *operador de captura* (&) que permite reificar (o "capturar") las declaraciones de función para que puedan ser utilizadas como expresiones.

2.1.2. Evaluación

Elixir es un lenguaje fuertemente tipado pero también de tipado dinámico, por lo que una de tareas del *runtime* consiste en realizar chequeos de tipo con el fin de prevenir la evaluación de operaciones ilegales. En caso de falla, los chequeos pueden lanzar errores, que son clasificados y enriquecidos con mensajes con el fin de facilitar su identificación.

Como lenguaje, Elixir se puede clasificar de tipo funcional, pues las estructuras de datos son inmutables además de que todas las funciones son *puras* por construcción: una vez invocadas se limitan a retornar con un valor resultado sin producir efectos colaterales en el contexto en que se evaluaron.

Comúnmente una expresión contendrá variables, así que el proceso de evaluación supone un contexto con los valores ligados a sus identificadores. Al ser evaluada, la expresión puede ligar tanto variables nuevas como volver a ligar las ya existentes; éstas se consolidan dentro de un nuevo contexto que será referido como el *contexto exportado por la evaluación*.

Para evaluar expresiones compuestas, aplican reglas específicas que determinan el orden en que las subexpresiones son evaluadas, así como para establecer el valor resultado. Además, para cada subexpresión existen reglas de alcance respecto al contexto exportado, que determinan si las modificaciones en éste último se consideran o no para la evaluación de las que le siguen.

Todo lo que en esta sección se presenta de manera informal, se reformula rigurosamente dentro de un modelo formal de evaluación en la Sección 4.2.

2.2. Especificación del fragmento de estudio

A continuación se introduce el fragmento de Elixir a partir del cual se desarrolla el sistema de tipos en el capítulo siguiente. La presentación de la sintaxis se realiza en EBNF.

2.3. Literales

Los literales representan a los valores de tipos primitivos del lenguaje. Para el fragmento se consideran únicamente números enteros (i), números flotantes (f), *strings* (c), y átomos (a). La sintaxis formal para su gramática es:

$$
\begin{array}{lcl}
l & ::= & i \mid f \mid a \\
i & ::= & [\texttt{-}]?[\texttt{0}-\texttt{9}]^+ \\
f & ::= & [\texttt{-}]?[\texttt{0}-\texttt{9}]^+\texttt{.}[\texttt{0}-\texttt{9}]^+ \\
c & ::= & \texttt{“}[\texttt{a}-\texttt{z},\texttt{A}-\texttt{Z},\texttt{0}-\texttt{9}]^*\texttt{”} \\
a & ::= & \texttt{:}[\texttt{a}-\texttt{z}][\texttt{a}-\texttt{z},\texttt{A}-\texttt{Z},\texttt{0}-\texttt{9}]^*
\end{array}
$$

Los números enteros se representan por cadenas de dígitos numéricos (`0`-`9`), en principio de largo arbitrario ya que se manejan internamente como enteros de máquina de precisión arbitraria. Para representar enteros negativos se prefija con un "`-`".

Los números flotantes se componen de dos cadenas de dígitos numéricos separados por el carácter ".". La representación de máquina es de 64-bit, pero para simplificar se asumira arbitraria y todas las cadenas serán admitidas. Para representar flotantes negativos se prefija con un "`-`".

Los strings son cadenas de largo arbitrario de caracteres Unicode, comenzando y terminando en comillas dobles. Para no complejizar la definición de la sintaxis, se consideran únicamente los alfanuméricos (`a`-`z`, `A`-`Z`).

Los átomos se identifican con cadenas de caracteres alfanuméricos prefijadas por el carácter ":" y comenzando en minúscula. Dentro de los átomos, se distinguen `:true`, `:false` y `:nil`, que el parser permite identificar aún sin prefijo como `true`, `false` y `nil`. Para simplificar la presentación se omitirán éstas variantes.

Algunos ejemplos de literales pueden ser:

```
42 # entero 42
42.0 # flotante 42
"atomo" # string "atomo"
:ejemplo # atomo "ejemplo"
:true # atomo "true"
```

Aunque sintácticamente son similares, los átomos no son un caso particular de los strings. Los primeros tienen pocas operaciones asociadas y están representados internamente como constantes, por lo que son ideales para utilizar en contextos en que se requiere un volúmen alto de comparaciones y se manejan pocas opciones, como claves en mapas o enumerados en una definición por casos.

2.4. Variables

Las variables se representan con identificadores formados por cadenas de caracteres alfabéticos y permiten hacer referencia a valores en un contexto de evaluación. Formalmente, la sintaxis para las variables es:

$$
\begin{array}{lcl}
id & ::= & [\texttt{a}-\texttt{z}][\texttt{a}-\texttt{z},\texttt{A}-\texttt{Z},\texttt{0}-\texttt{9},\texttt{\_}]^* \\
x & ::= & id
\end{array}
$$

En particular, de la definición se observa que los identificadores de las variables admiten el carácter "`_`" y también mayúsculas, aunque están obligadas a comenzar con una minúscula.

La convención del lenguaje es utilizar el estilo *snake case* para los identificadores; es decir, únicamente minúsculas y guión bajo como separador. Por ejemplo:

```
variable_en_snake_case
```

2.5. Expresiones

Como ocurre en la mayoría de los lenguajes de programación, las expresiones pueden utilizarse para definir variables, valores de tipos primitivos, estructuras de datos y llamadas a funciones. En Elixir, al igual que en Ruby, Erlang y varios otros lenguajes funcionales las expresiones ocupan un rol central en la sintaxis ya que también cubren a todas las estructuras de control existentes. Dentro del cuerpo de una declaración Elixir aplica el *motto* "todo es una expresión".

Como las expresiones pueden ligar variables, para describir la evaluación se deberá tomar en cuenta como afectan dichas ligaduras al contexto de variables exportado además de los valores finales y los posibles errores de ejecución.

2.5.1. Literales

Los literales -que ya fueron presentados como categoría sintáctica- son expresiones válidas del lenguaje. Su sintaxis se incluye en la gramática de las expresiones:

$$e ::= \ldots \mid l \mid \ldots$$

La evaluación de un literal es trivial, en cuanto otorga su propio valor y no modifica el contexto de variables.

```
42 # evalúa al entero 42
42.0 # evalúa al flotante 42
"42" evalúa al string "42"
:constante # evalúa al átomo "constante"
```

2.5.2. Variables

Las variables -que ya fueron presentadas como categoría sintáctica- son expresiones válidas del lenguaje. Su sintaxis se incluye en la gramática de las expresiones:

$$e ::= \ldots \mid x \mid \ldots$$

La evaluación de una variable otorga el valor ligado a su identificador en el contexto de variables.

```
variable_definida = 42 #
variable_definida # evalúa al entero 42
```

El proceso de compilación de Elixir incluye un análisis estático del código que detecta ocurrencias de variables que no están ligadas en su contexto correspondiente; así que es posible obviar ese escenario en código compilado (y por lo tanto, en el sistema de tipos que se presentará más adelante).

```
variable_no_definida # no compila
```

2.5.3. Operadores básicos

Los operadores básicos en Elixir se pueden utilizar sin necesidad de importar librerías; permiten realizar operaciones aritméticas, booleanas, y de comparación, entre otras. Para el fragmento de estudio, se incluyen:

- Operadores aritméticos unarios: `-`, `abs`
- Operadores booleanos unarios: `not`
- Operadores aritméticos binarios: `+`, `-`, `*`, `/`, `div`, `rem`
- Operadores booleanos binarios: `and`, `or`
- Operadores binarios en strings: `<>`, `==`
- Operadores de comparación binarios: `<`, `>`, `<=`, `>=`, `===`, `!==`, `max`, `min`

Salvo `max`, `min`, `abs`, `div` y `rem`, el resto de los operadores binarios incluidos en el fragmento se escriben de forma infija. Para simplificar, este hecho se omite en la presentación de la sintaxis EBNF que viene a continuación:

$$
\begin{array}{lll}
e & ::= & \ldots \mid \ominus e \mid e \oplus e \mid \ldots \\
\ominus & ::= & \texttt{-} \mid \texttt{abs} \mid \texttt{not} \\
\oplus & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{div} \mid \texttt{rem} \mid \texttt{and} \mid \texttt{or} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \\
 & & \texttt{<=} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{===} \mid \texttt{!==} \mid \texttt{max} \mid \texttt{min}
\end{array}
$$

Se debe tener en cuenta que cada operador tiene sentido únicamente para ciertos subconjuntos de valores -es decir, dentro de un dominio, si se piensa en su representación matemática-, y se incurrirá en errores de ejecución en caso de intentar operar con valores incorrectos para un operador dado.

La semántica de los operadores es la que se esperaría para la mayoría de los casos. A continuación algunos ejemplos y observaciones:

```
not :true # evalúa a :false
not 1 # ArgumentError
:true or :false # evalúa a :true
:true and 1 # evalúa a 1
:false and 1 # evalúa a :false
1 or :true # BadBooleanError
:nil or :true # BadBooleanError
```

Se observa:

- La negación está definida únicamente para argumentos booleanos.
- La disyunción está definida únicamente cuando el operando a la izquierda es booleano (lo mismo para la conjunción).
- Cuando ambos operandos son booleanos, la disyunción evalúa a un booleano (lo mismo para la conjunción).
- Cuando los operandos no tienen sentido para el operador en cuestión, se lanzan errores de tipo en tiempo de ejecución.

```
42 + 11 # evalúa al entero 53
42.0 + 11 # evalúa al flotante 53.0
42.0 + 11.0 # evalúa al flotante 53.0
42 + :true # lanza un ArithmeticError
```

```
42 / 0 # lanza un ArithmeticError
42 / 1 # evalúa al flotante 42
div(42, 2) # evalúa al entero 21
div(42, 2.0) # lanza un ArithmeticError
```

Se observa:

- La suma evalúa a un flotante si alguno de los operandos es flotante (lo mismo ocurre con la resta y el producto).
- La división siempre evalúa a un flotante.
- La suma está restringida para operandos numéricos, tanto enteros como flotantes (lo mismo ocurre con la resta y el producto).
- La división entera está restringida a operandos enteros (lo mismo ocurre con el operador de módulo `rem`).
- Cuando los operandos no tienen sentido para el operador en cuestión, se lanzan errores de tipo en tiempo de ejecución.

```
"a" == "bc" # evalúa a false
"bc" == "bc" # evalúa a true
11 <> "ab" # evalúa a ArgumentError
"ab" <> "cd" # evalúa al string "abcd"
"ab" <> 11 # evalúa a ArgumentError
```

Se observa:

- La igualdad de strings corresponde a la igualdad sintáctica.
- La concatenación de dos strings devuelve un string con el resultado de la concatenación de los operandos.
- Cuando alguno de los dos operandos de la concatenación no es un string, se lanzan errores de tipo en tiempo de ejecución.

```
1 > 1.0 # evalúa a :false
1.0 > 1 # evalúa a :false
1.2 > 1.0 # evalúa a :true
true > false # evalúa a :true
max(1.0, 1) # evalúa a 1.0
max(1, :a) # evalúa a :a
1 === 1.0 # evalúa a :false
1 === :a # evalúa a :false
```

Se observa:

- Los operadores de comparación son irrestrictos
- Los operadores de comparación asumen que los átomos son mayores a los números
- Las equivalencias distinguen tipos

Finalmente, es posible la existencia de *matchings* dentro de los operandos, como es el caso en la expresión `(x = 1) + (y = 2)`. En dichos casos, la evaluación resuelve primero las subexpresiones; en el primer caso pasando de `(x = 1)` a `1` y de `(y = 2)` a `2`. El resultado se compone tanto del valor final de la operación (`3`) como del contexto modificado por las asignaciones (`x` ligado a `1` e `y` ligado a `2`).

Un hecho a mencionar es que la evaluación de los operandos va de izquierda a derecha, por lo que si hubiera colisiones en las asignaciones, se prioriza el de más a la derecha para formar parte del contexto exportado. Así, en `(x = 1) + (x = 2)`, se exporta `x` con el valor `2`. A continuación se brindan más ejemplos:

```
abs(x = -1) # evalúa a 1 y x se exporta con el valor -1
(x = 1) + x # evalúa a 0, x se exporta con el valor 1
(x = 1) + (y = 2) # evalúa a 3, se exportan x con el valor 1 e y con el valor 2
(x = 1) + (x = 2) # evalúa a 3 y x se exporta con el valor 2
```

Se observa que en la segunda linea, `x` se exporta con el valor `1` en el primer argumento, y esto no afecta el contexto de evaluación del segundo, donde sigue valiendo `-1`. Esto explica por qué el resultado de la suma termina siendo `0`.

2.5.4. Estructuras de datos

Además de los literales de tipos primitivos, Elixir provee varios constructos que permiten declarar estructuras de datos con una sintaxis amigable. En el fragmento se incluyen los constructores para listas, tuplas y mapas; se omite el de *keyword lists*[1]. Además, se impone la restricción de las claves de mapas a literales, en vez de admitir expresiones arbitrarias.

Los constructores incluidos se pueden componer recursivamente para formar estructuras complejas.

$$e \quad ::= \quad e \mid \texttt{[]} \mid\mid \texttt{[}e\texttt{|}e\texttt{]} \mid \texttt{\{}e\texttt{,}\ldots\texttt{,}e\texttt{\}} \mid \texttt{\%\{}l\texttt{ => }e\texttt{,}\ldots\texttt{,}l\texttt{ => }e\texttt{\}}$$

En la definición EBNF, se incluyen:

- Constructores de listas: `[]` para la lista vacía y `[· | ·]` para construir una lista dada una expresión para la cabeza y otra para la cola.
- Constructores de tuplas: `{·,...,·}` para construir una tupla dadas las expresiones para sus coordenadas. Quedan implícitos los constructores para tuplas de tamaño 0 (`{}`) y 1 (`{·}`).
- Constructores de mapas: `%{· => ·,...,· => ·}` para un mapa dados literales para las claves y expresiones para sus valores. Quedan implícitos los constructores para mapas de tamaño 0 (`%{}`) y 1 (`%{· => ·}`).

La evaluación de las estructuras de datos se realiza de izquierda a derecha respecto a la ubicación de las subexpresiones; las precedencias para la consolidación del contexto exportado son en el mismo orden. Por ejemplo:

```
[1 | [2]] # lista ya evaluada
[x = 1 | [x = 2]] # evalúa a [1 | [2]] y x se exportan con el valor 2
{[1 | []], y = 2} # evalúa a {[1 | []],2}, y se exporta con el valor 2
%{:a => 1, :b => :true}} # mapa ya evaluado
%{:a => {x = 1, x = 2}} # evalúa a %{:a => {1, 2}}, x se exporta con el valor 2
```

Notar que no hay ninguna restricción para la cola de una lista. Por ejemplo, `[1 | [2 | vacio]]` es una lista válida cuya cola es una variable -que eventualmente podría ser reemplazada por otra lista-. Incluso se admiten casos como `[1 | [2 | :vacio]]`, cuya utilidad es más difícil de justificar y que será rechazado por el sistema de tipos, como se verá más adelante.

[1] Las keyword lists son listas de tuplas con ciertas restricciones de formación y se distinguen con una sintaxis especial. Presentan una API orientada a las operaciones de clave-valor.

2.5.5. *Pattern matching*

El *pattern matching* es una expresión distintiva de Elixir que permite ligar variables a valores que ocurren anidados dentro de estructuras de datos. La sintaxis se incluye en el de las expresiones como:

$$e \quad ::= \quad \ldots \mid p \ = \ e \mid \ldots$$

En el caso más simple, p toma la forma de una variable y el *pattern matching* funciona como un *binding* estándar. Es importante mencionar, sin embargo, que a diferencia de otros lenguajes compilados Elixir no provee un mecanismo explicito para inicializar variables, así que una expresión de asignación puede o bien estar ligando una nueva variable o bien religando una variable ya existente con un nuevo valor. Por ejemplo:

```
# se supone que el identificador x no está en el contexto
x = 1 # x asigna el valor 1 a x en el contexto
x = 1.0 # reasignando el valor de x en el contexto a 1.0
```

En realidad, las variables en el lado izquierdo constituyen un caso particular del uso de patrones. Los patrones son similares a las estructuras de datos, pero pueden contener porciones ignoradas con "_". Su propósito es "guiar" una asignación anidada. Por ejemplo:

- `{x, _}` identifica con `x` la primera componente de una tupla de dos elementos.
- `{x, 1}` identifica con `x` la primera componente de una tupla de dos elementos cuyo segundo elemento es el entero `1`.
- `[x | y]` identifica con `x` la cabeza e `y` con la cola de una lista no vacía.
- `%{:a => x,:b => _}` identifica con `x` al valor correspondiente a `:a` e ignora el correspondiente a `:b`.

Los patrones constituyen una categoría sintáctica independiente e incluyen otras variantes; se le dedica la Sección 2.6 con todos los detalles.

El *pattern matching* usando patrones compuestos resulta útil en combinación con estructuras de datos, donde pueden usarse para asignar en paralelo múltiples variables que ocurran anidadas dentro de éstos.

```
{x, _} = {:a, :b} # x el valor :a
{x, y} = {:a, :b} # x el valor :a, y toma el valor :b
[x | [y | _]] = [1 | [2 | [3]]]] # x, y toman los valores 1 y 2
%{:b => x} = %{:a => 1, :b => 2, :c => 3} # x toma el valor 2
```

Notar que la sintaxis especificada para el *pattern matching* no pone restricciones de la forma del patrón p con respecto a la de la expresión e; es responsabilidad del programador asegurarse que la expresión tomará un valor de la forma correcta al escribir el pattern match. La evaluación lanza un error `MatchError` si esto no es el caso.

```
{x, 1} = {1, 2} # MatchError
{x, _} = {1, 2, 3} # MatchError
{x, _} = una_lista # MatchError si una_lista es una lista
{x, _} = foo(:a, :b) # MatchError si foo(:a,:b) no devuelve una dupla
```

Formalmente, la evaluación de un *pattern matching* $p \ = \ e$ se puede considerar en tres etapas:

1. Se evalúa la expresión e a un valor v y exporta un contexto de evaluación.

2. Se intenta "matchear" el patrón p respecto al valor v[2]. O bien se tiene éxito, y en ese caso se genera un contexto con las variables del patrón, o falla.
3. En caso de falla del paso 2., se lanza un `MatchError`. Si no, el match evalúa a v y exporta un contexto que combina los del paso 1. y 2., con precedencia en el segundo.

El proceso de evaluación se puede ilustrar con el caso `{y,x} = {x = 1,2}`; aquí el *pattern matching* evalúa a `{1,2}` y se exportan `x` e `y` con valores `2` y `1` respectivamente.

2.5.6. Estructuras de control

En general, se llaman estructuras de control a los mecanismos del lenguaje que son capaces de alterar el flujo de ejecución de las instrucciones de un programa. En Elixir, hay varios de ellos, pero para el fragmento de estudio se consideran únicamente secuencias de expresiones, if/else y definición por casos. A continuación se presenta la porción de la sintaxis que corresponde a éstas estructuras:

$$
\begin{array}{rcl}
e & ::= & \ldots \\
 & | & (e\texttt{;})^{*}e \\
 & | & \texttt{if } e \texttt{ do } e \texttt{ else } e \texttt{ end} \\
 & | & \texttt{case } e \texttt{ do } (p \texttt{ => } e)^{+} \texttt{ end} \\
 & | & \ldots
\end{array}
$$

2.5.6.1. Secuencia

Permite escribir listas de expresiones para ser ejecutadas secuencialmente. Se puede escribir en la modalidad *inline* -que consiste en escribirlas una por una separadas por "`;`"- o bien separadas por saltos de linea. Para usar esta última variante anidada dentro de otra expresión, es necesario escribirlo entre paréntesis.

La evaluación de una secuencia va de arriba a abajo, y en cada caso el contexto exportado se combina con el contexto de evaluación original antes de evaluar la siguiente. El valor de toda la secuencia se identifica con el de la última expresión (la de más abajo). Un ejemplo de secuencia podría ser:

```
x = (z = 1); z + 2 # secuencia inline evalúa a 3, x y z se exportan con el valor 1
p = x + z # p toma valor 2
p + x + 3 # secuencia evalúa a 6
```

2.5.6.2. If/else

Funciona de manera similar a otros lenguajes de programación. Se puede escribir inline o con saltos de línea entre las dos ramas.

La evaluación es secuencial entre la condición y la rama seleccionada por la condición; la otra rama no se evalúa. A diferencia del caso de la secuencia, el contexto exportado es el de la condición; cualquier ligadura dentro de la rama evaluada no se considera para las evaluaciones siguientes.

```
x = :true
if (b = :true) do
      b and (x = :false)
else
```

[2]El proceso de evaluación de patrones respecto a un valor -informalmente referido como "*matching*"- se detalla en la Sección 2.6

```
    1 = 2
end # evalúa a :false
x # evalúa a :true
b # evalúa a :true
```

Notar que la ligadura de `x` en la rama seleccionada no se considera en el contexto exportado (ya que sigue valiendo `true`). Notar también que en la condición no se hizo una comparación de igualdad `b == :true`, sino que se ligó el valor `:true` a la variable `b`.

2.5.6.3. Definición por casos

Las definiciones por casos comienzan con la palabra reservada `case`. Se forman a partir de una expresión inicial y una lista ordenada de cláusulas formadas por pares (p, e) para un patrón p y una expresión e.

La evaluación comienza por la expresión inicial; el valor obtenido se intenta "matchear" respecto a los patrones en cada una de las cláusulas hasta que alguno sea exitoso y en dicho caso se prosigue a evaluar la expresión de la clausula correspondiente -con el contexto posiblemente modificado por el proceso de *matching*-. De manera similar al if/else, el contexto exportado considera únicamente la evaluación de la expresión testeada.

```
case {x = 0, 1} do
      {1, _} -> x
      {x, x} -> x*2
      {y, x} -> x*3 + y
end # evalúa a 3
```

A continuación se describe la evaluación del `case` para el ejemplo anterior:

1. La expresión testeada evalúa a `{0, 1}` ligando `x` al valor `1`.
2. El valor `{0, 1}` no matchea con el patrón `{x, x}`.
3. La evaluación del patrón `{x, y}` respecto a `{0, 1}` tiene éxito. Se liga `y` a `0`, se religa `x` a `1` y pasa a evaluarse la expresión de la derecha.

Modificando sutilmente el ejemplo anterior se puede lograr uno nuevo donde no se termina ejecutando ninguna de las ramas, lanzando un `CaseClauseError`:

```
case {x = 0, 1} do
      {1, _} -> x
      {x, x} -> x*2
      {y, x, _} -> x*3 + y
end # CaseClauseError
```

2.5.7. Operador de captura

Para introducir el alto orden el lenguaje provee dos mecanismos: declaraciones explicitas de funciones anónimas y operadores de captura. En el fragmento se incluye únicamente el segundo, que permite utilizar las funciones declaradas en el módulo al nivel de las expresiones. La sintaxis para éste es:

$$\begin{array}{rcl} f\_name & ::= & id \\ arity & ::= & [0-9]^{+} \\ e & ::= & \ldots \mid \&f\_name/arity \mid \ldots \end{array}$$

El operador de captura funciona anteponiendo un "&" al par (nombre, aridad) de una función declarada en el módulo -separados por un "/". La aridad resulta necesaria para la identificación porque Elixir admite la reutilización de nombres para declaraciones con distinta cantidad de parámetros.

A continuación se presenta un ejemplo, asumiendo que `sum` y `mult` de aridad dos están declaradas en el módulo.

```
sum_anon = &sum/2
mult_anon = &mult/2
sum_if_cond_else_mult_anon = if cond do
    sum_anon
else
    mult_anon
end
sum_if_cond_else_mult_anon.(2,3) # 5 cuando cond es true, sino 6
```

Finalmente, a efectos de la evaluación de patrones, la igualdad de declaraciones capturadas responde a la igualdad del par (nombre, aridad).

2.5.8. Llamadas a funciones

La sintaxis para invocar funciones distingue el caso en que la función en cuestión sea declarada o anónima. La sintaxis para ambas se incluye en la de las expresiones:

$$e \quad ::= \quad \ldots \mid f\_name(e,\ldots,e) \mid e.(e,\ldots,e) \mid \ldots$$

Notar que en el caso anónimo la invocación se escribe separando con un "." el nombre de la expresión respecto a la tupla de parámetros.

La evaluación se realiza de manera análoga a la definición por casos, considerando de una en una todas las instancias de declaración para dicho nombre de función y la aridad de los argumentos. En el caso anónimo, hay un paso previo que chequea que el valor referido por la variable sea una función y además de la aridad correcta.

Para los ejemplos a continuación, se asumen las declaraciones de `fetch/2` según el Recuadro 2.1:

```
fetch([:a | [:b | [:c | []]]], 1) # evalúa a :b
fetch(:a, 1) # FunctionClauseError
fetch([:a | [:b | []]], 2) # FunctionClauseError
fetch([:a | [:b | []]], {2}) # ArithmeticError
fetch([:a | [:b | []]], 1, 1) # no compila
```

Notar que en segundo caso, el error `FunctionClauseError` ocurre porque no se encuentra ningún caso donde los patrones evalúen exitosamente respecto a los argumentos. En el tercero, esto ocurre luego de dos llamadas recursivas, para los argumentos (`[]`, `0`).

Los siguientes ejemplos muestran varias instancias de invocaciones anónimas:

```
fetch_anon = &fetch/2
fetch_anon_attempt = :fetch_2
fetch_anon.([:a | [:b | [:c | []]]], 1) # evalúa a :a
fetch_anon.(:a, 1) # FunctionClauseError
fetch_anon.(:a, 1, 2) # BadArityError
fetch_anon_attempt.(:a, 1, 2) # BadFunctionError
```

```
def fetch([head | _], 0) do
    head
end

def fetch([_ | tail], n) do
   fetch(tail, n-1)
end
```

Recuadro 2.1: Declaraciones de fetch

2.6. Patrones

Los patrones constituyen una categoría sintáctica de términos que el lenguaje utiliza para varios propósitos: como lado izquierdo del pattern match, como parámetros de funciones y para las clausulas en las definiciones por casos. Formalmente la sintaxis es:

$$p \quad ::= \quad \_ \mid l \mid x \mid \hat{}x \mid [\,] \mid \mid [p \mid p] \mid \{p, \ldots, p\} \mid \%\{l \Rightarrow p, \ldots, l \Rightarrow p\}$$

En la sintaxis se incluyen el patrón *wild* (_), literales (l), variables (x) y variables "pinneadas" ($\hat{}x$), además de constructores de listas ([ ], $[p \mid p]$), de tuplas ($\{p, \ldots, p\}$) y de mapas ($\%\{k \Rightarrow p, \ldots, k \Rightarrow p\}$). Para simplificar el fragmento de estudio, se omiten dos variantes de la sintaxis completa: *bindings* de la forma $p = p$ y patrones con variables para las claves de mapas.

Los patrones pueden pensarse como predicados de estructura sobre los valores del lenguaje, y como tales pueden evaluarse respecto a un valor determinado y asignar sus variables en el proceso. Evitando una descripción formal, los siguientes ejemplos de *pattern matching* ilustran el mecanismo de evaluación (*matching*) de un patrón respecto a un valor:

```
1  _ = 42 # ok
2  42 = 42 # ok
3  42 = 42.0 # falla
4  x = 42 # ok y x toma el valor 42
5  ^x = 42 # ok
6  ^x = 42.0 # falla
```

A continuación se dan algunas explicaciones para los ejemplos presentados arriba:

1. Hay match porque el patrón _ admite cualquier valor
2. Hay match porque el patrón `42` refiere al mismo literal `42`
3. No hay match porque el patrón `42` no refiere al mismo literal `42.0`
4. Hay match y `x` asume el valor `42` en el contexto de evaluación
5. Hay match porque el valor de `x` en el contexto de evaluación (`42`) refiere al mismo valor `42`
6. No hay match porque el valor de `x` en el contexto de evaluación (`42`) no refiere al mismo valor `42.0`

```
[x | _] = [42 | [22 | true]] # éxito, y x toma el valor 42
[] = [42 | [22 | true]] # falla
{x, x} = {42, 43} # falla
{x, y, z} = {42, 43} # falla
{x, z} = {42, 43} # éxito, x toma el valor 42 y z toma el valor 43
%{:a => x} = %{:b => 3} # falla
%{:a => x} = %{:a => 2, :b => 3} # éxito, y x toma el valor 2
```

Los ejemplos de arriba involucran estructuras compuestas y muestran como el proceso de *matching* comienza asegurándose de que el valor de la derecha tenga la forma requerida por el patrón antes de matchear cada una de las componentes. Notar que en el caso de los mapas, es suficiente con que el valor de la derecha posea al menos las claves que indica el patrón.

El uso de variables "pinneadas" suele ser de utilidad dentro de definiciones por casos, como se ilustra en el siguiente ejemplo para la función `remove/2` que remueve todas las ocurrencias del primer argumento en la lista pasada como segundo argumento:

```
def remove(item, list) do
  case list do
    [^item | tail] -> remove(item, tail)
    [item | tail] -> [item | remove(item, tail)]
    [] -> []
  end
end
```

2.7. Declaraciones de función

Las declaraciones de función ocurren siempre a nivel de un módulo y comienzan con la palabra reservada `def`. Se identifican con un nombre que comienza en minúscula seguido de cualquier cantidad de caracteres alfanuméricos al igual que las variables.

Cada declaración se compone, además del nombre, por una lista de patrones para los parámetros y una expresión (en general dentro de una secuencia) para el cuerpo. La sintaxis queda en:

$$f \quad ::= \quad \texttt{def}\ f\_name(p, \ldots, p)\ \texttt{do}\ e\ \texttt{end}$$

Para invocar una función dentro de una expresión se utiliza la sintaxis estándar presentada en 2.5.8; los parámetros son evaluados secuencialmente respecto a las expresiones pasadas como argumentos y en caso de éxito genera el contexto para utilizar en la evaluación del cuerpo. Las modificaciones al contexto realizadas durante la evaluación no afectan de ninguna manera al contexto de invocación.

En un módulo se admiten múltiples declaraciones con el mismo nombre y cantidad de parámetros; éstas funcionan en conjunto como una definición por casos y cada invocación se considera en orden hasta encontrar la primera en que la evaluación de los parámetros tenga éxito.

A continuación se presenta como ejemplo la implementación del predicado `is_prefix` que toma dos expresiones y devuelve true cuando ambas son listas y además la primera es un prefijo de la segunda.

```
def is_prefix([head | tail_1], [head | tail_2]) do
    is_prefix(tail_1, tail_2)
```

```
end

def is_prefix([], []) do
    true
end

def is_prefix([], [_|_]) do
   true
end

def is_prefix(_, _) do
   false
end
```

Se observa:

- La primera declaración permite inspeccionar recursivamente listas que compartan el primer elemento y falla -pasando a la siguiente- en el resto.
- La segunda y tercera declaración devuelven `true` en los casos en que el primer argumento es una lista vacía y el segundo es una lista cualquiera.
- La cuarta declaración devuelven `false` en cualquier otro caso.

Notar que la correctitud depende del orden de las declaraciones; por ejemplo, si la última clausula se reubica al principio la función devuelve `false` en cualquier caso.

2.8. Declaración de módulo

Los módulos permiten agrupar funciones y constituyen la unidad básica de encapsulamiento de lógica en Elixir. En un módulo pueden ocurrir tanto declaraciones de función como de otros módulos, además de especificaciones de tipo para las funciones declaradas. En el fragmento de estudio se ignora la jerarquía de módulos, así que todo el código de un programa estará contenido dentro de un único módulo `Program`.

$$
\begin{array}{lcl}
s & ::= & \texttt{@spec}\ \ f\_name(T, \ldots, T)\ \ \texttt{::}\ \ T \\
d & ::= & f \mid s \\
p & ::= & \texttt{defmodule Program do}\ \ d\ \texttt{,}\ \ldots\texttt{,}\ \ d\ \texttt{end}
\end{array}
$$

Como se mostró en la sección anterior, una práctica común para el lenguaje consiste en declarar funciones a partir de múltiples instancias. Más generalmente, un módulo admite declaraciones con el mismo nombre y distinta aridad; éstas serán tratadas como funciones diferentes. Por ejemplo, en un módulo donde ocurran múltiples instancias de `foo` para uno y dos parámetros quedarán declaradas dos funciones `foo/1` y `foo/2` que agrupan respectivamente las instancias con cada aridad.

Las especificaciones de tipos (s) permiten agregar información de tipos a las signaturas de las declaraciones de función del módulo. Como tal, son completamente opcionales y se omiten al momento de la compilación; sin embargo, son incluidas por el parser dentro del AST, lo que hace posible el desarrollo de analizadores estáticos -como es el caso de *Dialyxir* basado en el concepto de *Success Typing* por Lindahl and Sagonas [2006], que permite encontrar errores en el código BEAM resultante de la compilación-. Las especificaciones son además, utilizadas en muchas librerias (incluidas las del propio lenguaje) como añadido de documentación.

Los tipos incluidos en las especificaciones deben respetar la sintaxis para los tipos de Elixir que se encuentra en la sección *Typespecs* de la documentación oficial. El capítulo siguiente presentará el diseño de un sistema de tipos basado en un fragmento de dicha sintaxis.

Capítulo 3

Elixir con tipos

Este capítulo propone un sistema de tipos algorítmico para el fragmento de Elixir que fue detallado en el capítulo anterior. La mayor parte de las reglas y conceptos involucrados se introducen en la primera sección. En la segunda sección se discuten los cambios y las extensiones necesarias para adaptar el sistema de tipos a una disciplina gradual.

Lo que se expone a continuación está basado fuertemente en Cassola et al. [2022]. La sintaxis de estudio, como podrá observarse, presenta varias simplificaciones, principalmente orientadas a evitar que las contribuciones del capítulo siguiente se diluyan en una presentación excesivamente larga.

A grandes rasgos, los aportes en este capítulo se pueden resumir en cuatro:

1. Modificaciones en los subsistemas de subtipado y de chequeo de expresiones que, se espera, permitirían garantizar la correctitud respecto a la semántica del lenguaje.

2. Cambios en el juicio de chequeo de patrones, que permiten validar más instancias y con tipos más informativos en algunos casos, sin comprometer la correctitud.

3. La presentación de un sistema algorítmico, que debería de poder ser llevado a una implementación concreta sin demasiados desafíos.

4. En la segunda sección, una extensión gradual algorítmica con un bosquejo de la prueba de la *Static Gradual Guarantee* (ver Siek et al. [2015]) respecto al sistema estático de la primera sección.

3.1. Sistema de tipos estático

A continuación se presenta un sistema de tipos para el fragmento de Elixir anteriormente descrito. A diferencia del sistema que se discute en la sección siguiente, éste se basa en una sintaxis de tipos y un conjunto de reglas "tradicionales" en el sentido de que cada tipo representa un conjunto concreto de los valores del lenguaje y las reglas hacen inferencias conservadoras en base a las expresiones analizadas y los tipos previamente otorgados.

Es importante mencionar que el sistema aquí presentado es de *chequeo* y no de *inferencia* y la información de tipos para ejecutar el análisis proviene de las especificaciones de tipo `@spec` existentes para todas las declaraciones del programa.

Tanto en esta sección como en la siguiente se consideran sistemas de *análisis estático*, que de ser implementados, chequean el código fuente sin modificarlo. Esto es un punto importante a mencionar ya que el capítulo

siguiente introduce un sistema de que define la inserción de casts dentro del código fuente a partir de chequeos exitosos, permitiendo trasladar parte de la información estática inferida a una mejora en la calidad del *runtime*.

Las reglas que conforman el sistema brindan información en forma de juicios, que a su vez dependen de la formulación rigurosa de la sintaxis para los tipos, las reglas del subtipado y los operadores de supremo e ínfimo. Todo esto será desarrollado en las secciones inmediatamente, tras lo cual se dará paso a la definición del conjunto de reglas.

3.1.1. Sintaxis de los tipos

Como fue dicho anteriormente, los tipos escogidos para definir el sistema corresponden únicamente a un fragmento estricto de la sintaxis completa que provee el lenguaje para los tipos básicos:

$$\begin{array}{lcl} B & ::= & \texttt{integer} \mid \texttt{float} \mid \texttt{number} \mid \texttt{string} \mid \texttt{atom} \mid \texttt{boolean} \mid a \\ T & ::= & B \mid \texttt{[ ]} \mid \texttt{[}T\texttt{]} \mid \texttt{\{}T\texttt{,} \ldots \texttt{,}T\texttt{\}} \mid \texttt{\%\{}l \texttt{ => } T\texttt{,} \ldots \texttt{,}l \texttt{ => } T\texttt{\}} \mid (T, .., T) \to T \end{array}$$

Dentro de los tipos básicos (B) se encuentran los enteros, flotantes, numéricos, átomos y booleanos. Los tipos `integer` y `float` sirven para representar valores literales enteros (i) y flotantes (f) respectivamente, mientras que `number` se usa para representar la unión de ambos. `atom` corresponde al conjunto de todos los átomos, y `boolean` únicamente a `:true` y `:false`. Además, se incluye en la sintaxis el símbolo de cada átomo a para representar el tipo singleton con ese valor.

Algunos ejemplos de valores y tipos básicos adecuados:

```
1 # tipa como integer y number
1.0 # tipa como float y number
"tres" # tipa como string
:atom # tipa como :atom y atom
:true # tipa como :true, :boolean y :atom
```

El resto de los tipos se definen por recursión sobre los constructores de listas, tuplas, mapas y funciones. Para listas se definen dos constructores de tipos `[ ]` y `[`T`]`, que representan a las lista vacía y posiblemente no vacía con elementos de tipo T. El constructor de tupla de aridad n sirve para representar tuplas con tipos adecuados para los valores de las coordenadas, y el constructor de mapa de aridad $2n$ representa a mapas donde hay *al menos* valores adecuados para los n tipos que corresponden a cada uno de los n literales apareados. Finalmente, el constructor de funciones de aridad $n+1$ sirve para representar funciones capturadas de aridad n dados tipos para los parámetros y el retorno en su especificación.

Algunos ejemplos de valores y tipos adecuados usando constructores:

```
[] # adecuado para [] y [T] para cualquier T
[1 | 2 | []] # adecuado para [integer] y [number]
[1 | 2.0 | []] # adecuado para [number]
{1, 2.0} # adecuado para {integer, float}, entre otros
%{:a => 1.0} # adecuado para %{:a => float}, %{:a => number} y %{}
%{:a => :a} # adecuado para %{:a => :a}, %{:a => atom} y %{}
&plus/2 # adecuado para (number, number) -> number
```

A diferencia de Cassola et al. [2022], aquí no se incluyen elementos en la sintaxis para representar el tipo vacío (allí `:none`) ni el tipo universal (allí `:term`). Esta decisión no compromete demasiado la expresividad del sistema estático, y por otra parte presenta ventajas para la construcción del sistema gradual en la sección siguiente.

3.1.2. Relación de subtipado

Si se piensa en los tipos como conjuntos de valores, el subtipado como relación de orden entre dos tipos corresponde a la relación de inclusión entre conjuntos de valores. Concretamente, un tipo τ se dirá *subtipo* de otro tipo σ - que se escribe $\tau \leq \sigma$-, si todos los valores de τ están también en σ.

La relación de subtipado que se elige para el sistema es una adaptación para el fragmento de un subtipado estructural estándar con *record types* (ver la Sección **5.** de Garcia et al. [2016], por ejemplo). La validez de la relación se presenta en un formato de reglas deductivas.

Para los tipos numéricos, hay dos reglas que establecen que los tipos `integer` y `float` son subtipos de `number`:

$$\frac{}{\texttt{integer} \leq \texttt{number}}\ \text{(ST\_INTEGER)} \qquad \frac{}{\texttt{float} \leq \texttt{number}}\ \text{(ST\_FLOAT)}$$

En Cassola et al. [2022] el subtipado para los numéricos se reducía a `integer` $\leq$ `float`. Ahora esto se modifica con el propósito de evitar la validación de instancias de *pattern matching* erróneas como `1 = 1.0`. Además, se introduce `number` para aprovechar las operaciones comunes a todos los valores numéricos (como se detalla en 3.1.8.3).

Para los átomos, hay tres reglas que reflejan las inclusiones previstas por la analogía conjuntista:

$$\frac{}{a \leq \texttt{atom}}\ \text{(ST\_ATOM)} \qquad \frac{a \in \{\texttt{:true}, \texttt{:false}\}}{a \leq \texttt{boolean}}\ \text{(ST\_BOOL)} \qquad \frac{}{\texttt{boolean} \leq \texttt{atom}}\ \text{(ST\_BOOLEAN)}$$

Finalmente, se añade la regla ST_B_REFL que decreta que cada tipo básico es subtipo de sí mismo (que según la analogía conjuntista corresponde a la noción de inclusión no es estricta):

$$\frac{\tau \in B}{\tau \leq \tau}\ \text{(ST\_B\_REFL)}$$

Para las listas, se declaran tres reglas que involucran al constructor del tipo de lista y el de lista vacía.

$$\frac{}{[\,] \leq [\,]}\ \text{(ST\_ELIST\_REFL)} \qquad \frac{}{[\,] \leq [\tau]}\ \text{(ST\_ELIST)} \qquad \frac{\tau \leq \sigma}{[\tau] \leq [\sigma]}\ \text{(ST\_LIST)}$$

La regla ST_ELIST_REFL establece que el subtipado es reflexivo para la lista vacía (notar que esto no es redundante con la regla ST_B_REFL porque ésta aplica únicamente para tipos en B). ST_ELIST establece que la lista vacía es también una lista de cualquier tipo y ST_LIST que el tipo de lista es un constructor covariante. Gracias a la covarianza es posible deducir, por ejemplo, que `[integer]` $\leq$ `[number]`.

Para tuplas hay una única regla que establece la covarianza del constructor de cualquier aridad en todas sus posiciones.

$$\frac{\tau_i \leq \sigma_i \quad \forall i \in \{1, ..., n\}}{\{\tau_1, ..., \tau_n\} \leq \{\sigma_1, ..., \sigma_n\}}\ \text{(ST\_TUPLE)}$$

Para funciones la regla establece que los parámetros son posiciones contravariantes y el retorno es covariante.

$$\frac{\begin{array}{c}\tau \leq \sigma \\ \sigma_i \leq \tau_i \quad \forall i \in \{1, ..., n\}\end{array}}{(\tau_1, .., \tau_n) \to \tau_0 \leq (\sigma_1, .., \sigma_n) \to \sigma_0}\ \text{(ST\_FUN)}$$

ST_FUN expresa que se puede ser más estricto en los tipos de los parámetros a la vez que relajar el tipo de retorno. Esto refleja formalmente el hecho que, por ejemplo, cualquier función de tipo `number` → `integer` puede ser utilizada como de tipo `float` → `number`.

Para definir las reglas del subtipado en mapas resulta conveniente definir la siguiente macro para un conjunto ordenado $I = (i_1, ..., i_n)$:

$$\texttt{\%\{}(\overline{k_i \texttt{ => } \tau_i})_{i \in I}\texttt{\}} ::= \texttt{\%\{}k_{i_1} \texttt{ => } \tau_{i_1}, ..., k_{i_n} \texttt{ => } \tau_{i_n}\texttt{\}} \qquad \text{(mapa como vector)}$$

Usando la notación mapa como vector, se puede definir las regla que para el subtipado entre mapas como:

$$\frac{(k_{i_1}, ..., k_{i_n}) = (l_1, ..., l_n) \text{ con } \{i_1, ..., i_n\} \subseteq I \qquad \tau_{k_{i_1}} \leq \sigma_{l_1} \quad \cdots \quad \tau_{k_{i_n}} \leq \sigma_{l_n}}{\texttt{\%\{}(\overline{k_i \texttt{ => } \tau_i})_{i \in I}\texttt{\}} \leq \texttt{\%\{}(\overline{l_j \texttt{ => } \sigma_j})_{i \in \{1,...,n\}}\texttt{\}}} \text{ (ST\_MAP)}$$

La regla ST_MAP es una formulación equivalente al de la Figura 16-2 de Pierce [2002] para la definición del subtipado algorítmico. Para generar intuición sobre ésta, es conveniente explicar tres casos particulares:

1. Las secuencias $(k_i)_{i \in I}$ y $(l_j)_{j \in J}$ son idénticas: la regla indica que los todas las posiciones de tipo dentro del constructor de mapas son covariantes. Notar la similitud con ST_TUPLE.

2. $I = \{i_1, ..., i_n\}$ y $\tau_{k_{i_j}} = \sigma_{k_j}$ $\forall j \in \{1, ..., n\}$: la regla indica que los dos mapas son idénticos a menos de la permutación $j \mapsto i_j$ si son escritos con la notación mapa como vector. Por ejemplo, siendo $I = \{i_1, i_2\}, (i_1, i_2) = (2, 1), (k_2, l_1) = (l_1, k_2) =$ `(:a, :b)` se tiene `%{:a => :t,:b => :u}` ≤ `%{:b => :u,:a => :t}` y viceversa.

3. $(l_j \texttt{ => } \sigma_j)_{j \in J}$ es un prefijo de $(k_j \texttt{ => } \tau_i)_{i \in I}$: Con esta restricción se logra plasmar el hecho de que un tipo de mapa dado también representa a todos los mapas con *al menos* las claves y valores del primero, sin excluir la posibilidad de que estos últimos tengan claves adicionales. Bajo esta semántica, tipos de mapa con más información representan menos valores; así que el subtipado va en el sentido opuesto al de la extensión. A modo de ejemplo, se tiene `%{:a => :t}` ≤ `%{}` y `%{:a => :t,:b => :u}` ≤ `%{:a => :t}`.

3.1.2.1. Propiedades

A continuación se explicitan algunas propiedades formales del subtipado que se tomarán en cuenta en el desarrollo posterior. En todos los casos, la prueba es muy sencilla y puede considerarse como parte del folclore.

Reflexivo: La regla ST_B_REFL únicamente declara que el subtipado es reflexivo cuando se restringe a tipos básicos. Pero de hecho, con un argumento inductivo simple es posible extender la reflexividad para cualquier tipo. Concretamente, se prueba que:

$$\frac{}{\tau \leq \tau} \text{ (ST\_REFL)}$$

es una regla admisible

Transitivo: Con un argumento inductivo se prueba que la siguiente regla es admisible:

$$\frac{\tau \leq \sigma \quad \sigma \leq \mu}{\tau \leq \mu} \text{ (ST\_TRANS)}$$

Preorden: Las dos propiedades anteriores determinan que el subtipado es un preorden. En realidad, la única regla que impide que el subtipado sea un orden es ST_MAP, ya que ésta permite identificar mapas a menos de una permutación de sus elementos.

Sin máximo ni mínimo: A diferencia de la presentación de Cassola et al. [2022], en este trabajo se considera un sistema de tipos sin máximo ni mínimo (allí `:term` y `:none`), aunque si hay elementos maximales y minimales. Por ejemplo `atom` como `number` son tipos maximales, mientras que `:true` e `integer` son minimales.

3.1.2.2. Subtipado algorítmico

Las reglas presentadas para el subtipado definen un sistema algorítmico, pues inducen de manera directa un algoritmo de decisión para corroborar $\tau \leq \sigma$ para todo par $(\tau, \sigma) \in T \times T$.

En efecto, se puede observar que cada par de tipos en una desigualdad aplica como conclusión en *a lo sumo* una regla, y en todas las reglas las hipótesis contienen únicamente variables que ya aparecen en la conclusión. Estos dos hechos motivan un algoritmo sencillo y sin backtracking, que es correcto y completo respecto al sistema de reglas. En pseudocódigo, el algoritmo Subtype se puede escribir como:

```
procedure Subtype(τ, σ)
    for (Hipotesis, Conclusion) regla del sistema do
        Σ ← Unificar(Conclusion, τ ≤ σ)
        if Σ = fail then
            continue
        end if
        IsSubtype ← true
        for H ← Hipotesis do
            IsSubtype ← IsSubtype and Valid(Subst(H, Σ))
        end for
        return IsSubtype
    end for
    return false
end procedure
```

teniendo en cuenta que :

- Unificar(Conclusion, $\tau \leq \sigma$) genera un contexto de tipos Σ en caso de lograr instanciar la fórmula $\tau \leq \sigma$ en la conclusión de la regla. De lo contrario falla.
- Subst(H, Σ) instancia las variables de una hipótesis H con los tipos en Σ.
- Valid(H) chequea la validez de una hipótesis H sin variables de tipo libres. En caso de que H sea de la forma $\tau \leq \sigma$, se llama de manera mutuamente recursiva a Subtype(τ, σ).

En realidad, el algoritmo para el subtipado puede definirse de modo mucho más performante y explícito sin hacer alusión a las reglas, similar a la Figura 3 de Garcia et al. [2016]. Este algoritmo se detalla en el Anexo B.2.

3.1.3. Operadores de supremo e ínfimo

Al ser un preorden, la definición de supremo e ínfimo tiene sentido para el subtipado a menos de identificar tipos equivalentes; es decir, a menos de permutaciones de los elementos en los mapas. A partir de ahora, esta

identificación se tomará como implícita, y se podrá considerar al subtipado como una relación de orden usual.

Se utilizarán las notaciones $\tau \vee \sigma$ y $\tau \wedge \sigma$ para referir al supremo e ínfimo binario respectivamente para dos tipos τ, σ. En caso de un conjunto arbitrario de tipos $\mathcal{T}$, se usan las notaciones $\bigvee \mathcal{T}$ y $\bigwedge \mathcal{T}$.

En el Recuadro 3.1 se define en pesudocódigo un algoritmo Sup para los operadores de supremo e ínfimo binario. Instanciado con $b =$ **true**, el algoritmo calcula el supremo, mientras que con $b =$ **false** calcula el ínfimo de dos tipos τ, σ. Como en este sistema no existen ni máximo ni mínimo, ambos son en realidad operadores parciales, y el algoritmo falla en caso de indefinición.

La correctitud del algoritmo puede garantizarse con una prueba formal, aunque ésto se ha dejado fuera del alcance. Informalmente, sin embargo, se le puede atribuir un sentido desde la perspectiva conjuntista: el supremo equivale [1] a la unión y el ínfimo a la intersección de dos conjuntos. Algunos ejemplos:

- `integer` $\vee$ `float` = `number` porque `number` cubre los casos de valores enteros y flotantes.
- `integer` $\wedge$ `float` no está definido porque `integer` y `float` no tienen elementos en común, y el conjunto vacío no tiene un correspondiente desde la sintaxis de tipos[2].
- `:a` $\vee$ `:b` = `atom` porque `atom`es la mejor aproximación por encima de `:a` y `:b` en la sintaxis de tipos.
- `{integer,atom}` $\vee$ `{float,:a}` = `{number,atom}` ya que las tuplas en ambos conjuntos tienen primera coordenada numérica y segunda de tipo átomo.
- `integer` $\to$ `atom` $\wedge$ `float` $\to$ `boolean` = `number` $\to$ `boolean` ya que todas las funciones incluidas aceptan todos los numéricos y devuelven en particular un booleano.
- `%{:age => integer}` $\wedge$ `%{:name => string}` = `%{:age => integer,:name => string}` ya que mapas de que cumplan con los dos tipos poseen ambas claves con los tipos correspondientes.
- `%{:age => integer}` $\vee$ `%{:name => string}` = `%{}` ya que el tipo `%{}` corresponde al conjunto de todos los mapas, del cual los conjuntos de mapas denotados por los tipos `%{:age => integer}` y `%{:name => string}` son subconjuntos (y no hay ningún conjunto más pequeño desde la sintaxis de tipos que los contenga a ambos)

Finalmente, para calcular supremos e ínfimos arbitrarios basta con iterar la definición binaria. A modo de ejemplo: $\bigvee\{\tau_1, \tau_2, \tau_3\} = (\tau_1 \vee \tau_2) \vee \tau_3$.

3.1.4. Juicios del sistema de tipos

A continuación se presentan los juicios que componen el sistema de chequeo. En las siguientes secciones se detallan las reglas correspondientes a cada uno de ellos, acompañando con ejemplos.

3.1.4.1. Chequeo de programa

El siguiente juicio indica que un programa m ha sido correctamente chequeado respecto al sistema de tipos:

$$\vdash^t m \qquad \text{(j. chequeo de programa)}$$

donde m es un módulo de la sintaxis del fragmento de Elixir.

[1] En realidad, esto es cierto a menos utilizar las aproximaciones dadas por la interpretación conjuntista de tipos a valores $\tau \mapsto V^\tau$, que es el adjunto superior de una *Conexión de Galois* respecto a una interpretación conjuntista para los tipos, como por ejemplo la de Frisch et al. [2002]

[2] En vista de la *Conexión de Galois*, es más preciso decir que no tiene un mejor aproximante desde la sintaxis cuando es visto como conjunto

procedure SUP(τ, σ, b)
  switch (τ, σ, b) **do**
    case ($\tau, \sigma,$ **true**) **when** SUBTYPE(τ, σ)
      return σ
    case ($\tau, \sigma,$ **false**) **when** SUBTYPE(τ, σ)
      return τ
    case (`integer`, `float`, **true**) **or** (`float`, `integer`, **true**)
      return `number`
    case ($a_1, a_2,$ **true**) **when** $a_1, a_2 \in$ {`:true`, `:false`}
      return `boolean`
    case ($a_1, a_2,$ **true**)
      return `atom`
    case (`[`τ`]`, `[`σ`]`, b)
      return `[`SUP(τ, σ, b)`]`
    case ({$\tau_1, \ldots, \tau_n$}, {$\sigma_1, \ldots, \sigma_n$}, b)
      return {SUP(τ_1, σ_1, b), ..., SUP(τ_n, σ_n, b)}
    case (`%{`$\overline{(k_i \Rightarrow \tau_i)_{i\in I}}$`}`, `%{`$\overline{(k_i \Rightarrow \sigma_i)_{i\in J}}$`}`, **true**)
      return `%{`$\overline{(k_i \Rightarrow \text{SUP}(\tau_i, \sigma_i, b))_{i\in I\cap J}}$`}`
    case (`%{`$\overline{(k_i \Rightarrow \tau_i)_{i\in I}}$`}`, `%{`$\overline{(k_i \Rightarrow \sigma_i)_{i\in J}}$`}`, **false**)
      return `%{`$\overline{(k_i \Rightarrow \text{SUP}(\tau_i, \sigma_i, b))_{i\in I\cap J}}, \overline{(k_i \Rightarrow \tau_i)_{i\in I\setminus J}}, \overline{(k_i \Rightarrow \sigma_i)_{i\in J\setminus I}}$`}`
    case ($(\tau_1, .., \tau_n) \to \tau_0, (\sigma_1, .., \sigma_n) \to \sigma_0, b$)
      return (SUP($\tau_1, \sigma_1,$ **not** b), .., SUP($\tau_n, \sigma_n,$ **not** b)) $\to$ SUP(τ_0, σ_0, b)
    default
      fail
end procedure

Recuadro 3.1: Algoritmo de supremo e ínfimo

3.1.4.2. Colección de especificaciones

El siguiente juicio expresa que Δ es el contexto de especificaciones formado al recolectar la información de todas las especificaciones del programa:

$$\vdash^{c} m \Rightarrow \Delta \qquad \text{(j. colección de especificaciones en programa)}$$

siendo Δ un contexto que asigna una firma a cada par (nombre de función, aridad). Formalmente se puede decir que $\Delta \in \textsc{Map}(\textsc{Id} \times \mathbb{N}, \textsc{List}(T) \times T)$.

Para cada declaración d del módulo, la colección individual de una nueva especificación (en caso que lo sea) enriquece el contexto Δ, reflejandolo en Δ':

$$\Delta \vdash^{c} d \Rightarrow \Delta' \qquad \text{(j. colección de especificaciones en declaración)}$$

3.1.4.3. Chequeo de programa

El siguiente juicio indica que un programa m ha sido correctamente verificado en el sistema de tipos respecto al contexto de especificaciones Δ:

$$\Delta \vdash^{ts} m \qquad \text{(j. chequeo de programa bajo especificaciones)}$$

donde m es un módulo de la sintaxis del fragmento de Elixir.

3.1.4.4. Chequeo de declaración

Para decir que una declaración d ha sido chequeada correctamente respecto a un contexto de especificaciones Δ, se define

$$\Delta \vdash^{t} d \qquad \text{(j. chequeo de declaración)}$$

3.1.4.5. Chequeo de expresión

Para indicar que una expresión e ha sido chequeada correctamente bajo un contexto de especificaciones Δ y uno de variables Γ, otorgando un tipo τ y exportando el contexto de variables asignadas en la expresión (Γ'), se define

$$\Delta; \Gamma \vdash^{t} e : \tau \Rightarrow \Gamma' \qquad \text{(j. chequeo de expresión)}$$

Aquí $\tau \in T$ es un tipo de la sintaxis y $\Gamma, \Gamma' \in \textsc{Map}(\textsc{Id}, T), \Delta \in \textsc{Map}(\textsc{Id} \times \mathbb{N}, \textsc{List}(T) \times T)$ son contextos de tipos para variables y especificaciones respectivamente.

Puede notarse la diferencia con Cassola et al. [2022], que trata a Γ' como el contexto actualizado; ahora eso se modifica para evitar un problema que sufrían las reglas de chequeo de tuplas y mapas respecto a la formación correcta de dicho contexto.

3.1.4.6. Chequeo-refinamiento de patrones

Recordar de la Sección 2.6 que la evaluación de patrones se define respecto a un valor v y el contexto de variables, y que de ser exitoso, devuelve como *output* el contexto de variables que fueron asignadas dentro de p. Si además se incluye a v dentro del output, puede decirse informalmente que la evaluación del patrón "tipa" como:

$$\textsc{Map}(Id, V) \times V \times P \rightharpoonup V \times \textsc{Map}(Id, V)$$

y que en caso de evaluaciones exitosas puede además restringirse a funciones totales[3]:

$$\textsc{Map}(Id, V) \times V \times P \to V \times \textsc{Map}(Id, V)$$

El juicio para el chequeo de patrones puede verse como el predicado estático correspondiente al proceso de evaluación de patrones, por lo que, de acuerdo a la analogía de tipos como conjuntos de valores, éste "tipa" según:

$$\textsc{Map}(Id, T) \times T \times P \to T \times \textsc{Map}(Id, T)$$

Esto último permite justificar la forma del juicio

$$\Sigma; \tau \vdash^{pt} p : \sigma \Rightarrow \Gamma \qquad \text{(j. chequeo-refinamiento de patrones)}$$

donde $\Sigma \in \textsc{Map}(Id, T)$, $\tau \in T$ y $p \in P$ funcionan como *input* al juicio y $\sigma \in T$ y Γ funcionan como *output*.

A diferencia de Cassola et al. [2022], en este trabajo el juicio de chequeo es algorítmico y como tal está obligado a especificar la modalidad *input/output* de cada componente. A priori, lo más directo es tratar al tipo τ como *input* (en un rol de tipo candidato), pero al asumir un *matching* exitoso, de las reglas del sistema se extrae nueva información que permite aprovecharlo tambi en la modalidad *output* (σ) como un refinado del *input*. De hecho, a posteriori será posible afirmar que $\sigma \leq \tau$, lo que permite hablar de σ como un *refinamiento* de τ.

Para el chequeo de un patrón, se precisan dos contextos: uno *input* (Σ) para los tipos de las variables "pinneadas" y otro *output* (Γ) para los de las variables asignadas dentro del patrón p.

Para la definición del juicio principal, se utilizan dos juicios auxiliares:

$$\Sigma; \Gamma; \tau \vdash^{pc} p \Rightarrow \Gamma' \qquad \text{(j. colección de variables en patrones)}$$
$$\Sigma; \Gamma; \tau \vdash^{ptv} p : \sigma \qquad \text{(j. chequeo de patrones bajo contexto de variables)}$$

El juicio de colección predica sobre la ligadura de los tipos de las variables en p dentro de un *matching* exitoso respecto al tipo candidato τ. Por otro lado, en el juicio de tipado se construye un tipo nuevo (σ) a partir de un contexto (ya recolectado) (Γ) para las variables del patrón p y el tipo candidato τ.

3.1.5. Reglas para el chequeo de un programa

El chequeo del módulo programa se hace en dos etapas: primero se recolectan las especificaciones en un contexto Δ y luego se chequea el módulo bajo ese contexto. Esto se representa en la regla:

$$\frac{\vdash^{c} m \Rightarrow \Delta \qquad \Delta \vdash^{ts} m}{\vdash^{t} m} \text{ (T\_PROG)}$$

La regla T_PROG se puede describir en tres etapas:

[3] Notar la diferencia entre el símbolo $\rightharpoonup$ y $\to$

1. Con el juicio j. colección de especificaciones en programa se indica que Δ contiene la información de todas las especificaciones de m.
2. Con el juicio j. chequeo de programa bajo especificaciones se indica que bajo las especificaciones en Δ, m se chequea correctamente.
3. Se concluye que juicio j. chequeo de programa es válido para m, es decir que m es un programa correcto dentro del sistema de tipos.

3.1.6. Reglas para la colección de especificaciones

La colección de especificaciones en el módulo se realiza iterando sobre todas las declaraciones que la conforman y añadiendo en un contexto cada especificación de acuerdo al par (nombre, aridad) que le corresponde. La iteración parte del contexto vacío, que por abuso de notación se escribe $\emptyset$.

$$\frac{\emptyset \vdash^c d_1 \Rightarrow \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash^c d_n \Rightarrow \Delta_n}{\vdash^c \texttt{defmodule Program do } d_1\texttt{, } \ldots\texttt{, } d_n \texttt{ end} \Rightarrow \Delta_n} \text{(COLLECT\_PROG)}$$

En cada declaración, la reglas COLLECT_DEF y ECOLLECT_SPEC distinguen si es de función (f) o de especificación (s), para modificar el contexto únicamente en el segundo caso.

$$\frac{}{\Delta \vdash^c f \Rightarrow \Delta} \text{(COLLECT\_DEF)}$$

$$\frac{(f\_name, n) \notin \text{dom}(\Delta)}{\Delta \vdash^c \texttt{@spec } f\_name(\tau_1, \ldots, \tau_n) \texttt{ :: } \tau_0 \Rightarrow \Delta[(f\_name, n) \mapsto ((\tau_1, \ldots, \tau_n), \tau_0)]} \text{(ECOLLECT\_SPEC)}$$

Notar que las reglas son completamente indiferentes al orden en que se presentan las declaraciones. La única restricción está dada por la condición $(f\_name, n) \notin \text{dom}(\Delta)$ en ECOLLECT_SPEC, ya que impide la posibilidad de tener de múltiples especificaciones para una misma signatura.

3.1.7. Reglas para el chequeo de un programa

Una vez que se consigue el contexto de especificaciones Δ, el chequeo del programa se realiza iterando sobre todas las funciones y chequeando cada una de ellas bajo Δ.

$$\frac{\Delta \vdash^t d_1 \quad \ldots \quad \Delta \vdash^t d_n}{\Delta \vdash^{ts} \texttt{defmodule Program do } d_1\texttt{, } \ldots\texttt{, } d_n \texttt{ end}} \text{(T\_PROG)}$$

Las reglas de chequeo para las declaraciones son:

$$\frac{}{\Delta \vdash^t s} \text{(T\_SPEC)} \qquad \frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \emptyset; \emptyset; \tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset; \Gamma_{n-1}; \tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta; \Gamma_n \vdash^t e : \sigma \Rightarrow \Gamma' \quad \sigma \leq \tau_0\end{array}}{\Delta \vdash^t \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end}} \text{(T\_DEF)}$$

La regla T_SPEC indica, de manera análoga a S_DEF, que las especificaciones son ignoradas al momento de realizar el chequeo.

T_DEF indica las etapas que deben seguirse para chequear una declaración de función. Leídos de arriba abajo e izquierda a derecha, uno por uno:

1. El par (f_name, n) debe aparecer como clave en Δ. Por construcción, su valor será de la forma $((\tau_1, \ldots, \tau_n), \tau_0)$, donde $\tau_1, \ldots, \tau_n$ son los tipos especificados para los parámetros y τ_0 para el cuerpo de la función.

2. Se iteran los parámetros p_i, uno a uno junto al tipo τ_i que corresponde por su posición para recolectar un contexto de variables Γ_n. En todos los casos el contexto externo es vacío, así que se omite con $\emptyset$.

3. El cuerpo e de la función se chequea bajo el contexto de especificaciones Δ y el contexto de variables Γ_n de los parámetros. El chequeo exitoso devuelve un tipo σ, que debe ser subtipo del especificado (τ_0).

Para ilustrar el proceso de chequeo, se propone una serie de ejemplos en base a la declaración de una función trivial `assert_is_double` que chequea que en una tupla la primera componente es el doble que la segunda:

Ejemplo 1:

```
def assert_is_double({x, y, _}) do
    x == :dos * y
end
```

Como la declaración no fue especificada, el chequeo falla en la primera etapa.

Ejemplo 2:

```
@spec assert_is_double({number, number}) :: number
def equals_x_differs_y({x, y, _}) do
    x == :dos * y
end
```

Notar que el chequeo falla en la segunda etapa porque el tipo del argumento no tiene la forma del patrón parámetro, que es una tupla de tres componentes en vez de dos.

Ejemplo 3:

```
@spec assert_is_double({number, number}) :: number
def equals_x_differs_y({x, y}) do
    x == :dos * y
end
```

El chequeo falla en el análisis del cuerpo (etapa 3), ya que `:dos` es un átomo y no un número.

Ejemplo 4:

```
@spec assert_is_double({number, number}) :: number
def equals_x_differs_y({x, y}) do
    x == 2 * y
end
```

El chequeo falla al comprobar que el tipo de retorno (`boolean`) no es subtipo de `number` (etapa 3).

Ejemplo 5:

```elixir
@spec assert_is_double({number, number}) :: boolean
def equals_x_differs_y({x, y}) do
    x == 2 * y
end
```

El chequeo termina exitosamente.

3.1.8. Reglas para el chequeo de una expresión

El juicio j. chequeo de expresión expresa el chequeo exitoso de una expresión e bajo un contexto de funciones Δ y uno de variables Γ, entregando un tipo τ y exportando un contexto de variables Γ'. A efectos de la definición de un sistema algorítmico -y siguiendo la terminología adoptada en Cimini and Siek [2016]-, tanto los contextos Γ, Δ y la expresión e deben leerse como *input* del juicio, mientras que τ y Γ' funcionan como *output*.

El tipo τ que entrega un chequeo exitoso representa, informalmente hablando, el menor tipo posible para e bajo Δ y Γ; y cualquier σ cumpliendo $\tau \leq \sigma$ se considera un tipo válido. Es posible formalizar esto último desarrollando, paralelamente al sistema algorítmico, un sistema declarativo como el de Cassola et al. [2022]. La formalización de este hecho sería muy similar al del Teorema 16.2.5 de Pierce [2002].

3.1.8.1. Chequeo de literales

Para los literales, hay una única regla que define el chequeo:

$$\frac{}{\Delta; \Gamma \vdash^t l : type(l) \Rightarrow \emptyset} \text{(T\_LIT)}$$

siendo *type* una función que asigna el tipo más preciso para cada literal. Notar que según la regla T_LIT, los literales siempre chequean correctamente con este tipo, y sin exportar variables.

El siguiente procedimiento describe, en pseudocódigo, la definición de *type*:

procedure TYPE(l)
  switch l **do**
    case i **when** ISINTEGER(i) : **return** `integer`
    case f **when** ISFLOAT(f) : **return** `float`
    case a **when** ISATOM(a) : **return** a
    case a **when** ISSTRING(c) : **return** `string`
end procedure

asumiendo que los procedimientos auxiliares ISINTEGER, ISFLOAT, ISATOM, ISSTRING devuelven **true** cuando el argumento respecta el *regex* especificado para los distintos tipos de literales en 2.3.

3.1.8.2. Chequeo de variables

La regla para las variables es:

$$\frac{\Gamma[x] = \tau}{\Delta; \Gamma \vdash^t x : \tau \Rightarrow \emptyset} \text{(T\_VAR)}$$

indicando que cada variable se chequea exitosamente con el tipo que tiene el contexto. Mientras que en Cassola et al. [2022] esta regla exporta Γ, ahora éste se reserva para variables explícitamente ligadas dentro de un patrón, por lo que en T_VAR se exporta el contexto vacío (∅).

3.1.8.3. Chequeo de operadores básicos

Para motivar la definición de las reglas de chequeo para los operadores, se comienza con algunos ejemplos de ejecución de la suma:

```
1 + 2 # evalúa a 3
1 + 2.0 # evalúa a 3.0
1.0 + 2 # evalúa a 3.0
1.0 + 2.0 # evalúa a 3.0
```

Del ejemplo se pueden deducir las siguientes observaciones:

1. Si los dos argumentos son de tipo `integer`, el resultado es de tipo `integer`
2. Si alguno de los argumentos es de tipo `float`, el resultado es de tipo `float`.
3. Si alguno de los argumentos es de tipo `number` y el otro es de tipo `integer`, el resultado es de tipo `number`, ya que podría ser tanto `integer` como `float`.
4. Si los dos argumentos son de tipo `number`, el resultado es de tipo `number`, ya que podría ser tanto `integer` como `float`.

Es posible reunir las observaciones anteriores dentro de un procedimiento que asigna el tipo para el resultado de la suma dados tipos para sus argumentos:

procedure SumType(τ, σ)
  switch (τ, σ) **do**
    case (`integer`, `integer`): **return integer**
    case (τ, `float`) **when** $\tau \leq$ `number` : **return float**
    case (`float`, σ) **when** $\sigma \leq$ `number` : **return float**
    case (τ, σ) **when** $\tau \leq$ `number` **and** $\sigma \leq$ `number` : **return number**
    default fail
end procedure

Notar que el procedimiento para *sumType* falla en caso de que alguno de los tipos no sea numérico. Esto es coherente con el hecho de que la evaluación de Elixir lanza un error `ArithmeticError` para estos casos:

```
1 + :true # lanza un ArithmeticError
:true + 2.0 #  lanza un ArithmeticError
:true + :true # lanza un ArithmeticError
```

La función *binOpType* que se encuentra en el Anexo B.4 generaliza a *sumType* para todos los operadores binarios en un único procedimiento que depende de un parámetro extra para el símbolo de operación binaria $\oplus$. Así, por ejemplo, $binOpType(+, \cdot, \cdot)$ corresponde a $sumType(\cdot, \cdot)$.

La regla para el tipado de los operadores binarios se define como:

$$\frac{\Delta; \Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta; \Gamma \vdash^t e_2 : \tau_2 \Rightarrow \Gamma_2 \qquad binOpType(\oplus, \tau_1, \tau_2) = \sigma}{\Delta; \Gamma \vdash^t e_1 \oplus e_2 : \sigma \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{(T\_BIN\_OP)}$$

donde la combinación de dos contextos Γ_1 y Γ_2 (con precedencia en Γ_2) se define por:

$$\begin{aligned}(\Gamma_1 \dagger \Gamma_2)[x] &= \Gamma_2[x] \quad \text{si } x \in \text{dom}(\Gamma_2)\\ (\Gamma_1 \dagger \Gamma_2)[x] &= \Gamma_1[x] \quad \text{si } x \notin \text{dom}(\Gamma_2)\end{aligned} \qquad \text{(combinación de contextos)}$$

La manipulación de los contextos de tipos en T_BIN_OP viene sin sorpresa si se toma en cuenta lo dicho en los últimos párrafos de la Sección 2.5.3 acerca de las modificaciones del contexto de evaluación en la ejecución de las operaciones binarias. Ambos operandos toman el contexto previo a la evaluación (que corresponde al contexto de tipos Γ) y lo modifican de manera independiente (corresponden a Γ_1 y Γ_2 respectivamente). El contexto que se exporta surge de combinar las modificaciones en ambos, dando precedencia al segundo operando (corresponde a $\Gamma_1 \dagger \Gamma_2$).

La regla T_UNI_OP para los operadores unarios es una versión simplificada de T_BIN_OP que depende de la definición *uniOpType* (también en el Anexo B.4).

$$\frac{\Delta; \Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \mathit{uniOpType}(\ominus, \tau) = \sigma}{\Delta; \Gamma \vdash^t \ominus e : \sigma \Rightarrow \Gamma'} \text{ (T\_UNI\_OP)}$$

3.1.8.3.1 Tipado de operadores en abstracto

En el anexo se encuentra la definición exhaustiva del *binOpType* y *uniOpType* para los operadores concretos que aparecen en la sintaxis del fragmento. Es de interés, sin embargo, explicitar ciertas propiedades de estas funciones que serán las utilizadas en el resto del trabajo.

A continuación se enuncian cuatro propiedades que se consideran esenciales para la función parcial $\mathit{binOpType} : B \times B \rightharpoonup B$; los enunciados se pueden generalizar para $\mathit{uniOpType} : B \rightharpoonup B$ sin dificultad:

1. **Correctitud**: Si l_1, l_2 son dos literales tales que $l_1 \oplus l_2 = l$ y $\tau_1, \tau_2 \in T$ son tales que $\mathit{type}(l_1) \leq \tau_1$ y $\mathit{type}(l_2) \leq \tau_2$, entonces $\mathit{type}(l) \leq \mathit{binOpType}(\oplus, \tau_1, \tau_2)$ siempre que este último esté definido.

2. **Monotonía**: Si $\tau_1, \tau_2, \sigma_1, \sigma_2 \in T$ son tales que $\tau_1 \leq \sigma_1$, $\tau_2 \leq \sigma_2$ y $\mathit{binOpType}(\oplus, \sigma_1, \sigma_2)$ está definido, entonces $\mathit{binOpType}(\oplus, \tau_1, \tau_2)$ está definido y además $\mathit{binOpType}(\oplus, \tau_1, \tau_2) \leq \mathit{binOpType}(\oplus, \sigma_1, \sigma_2)$.

3. **Argumento máximo**: Para cada $\oplus$, existen $\tau_1^{\oplus}, \tau_2^{\oplus} \in T$ tales que $\mathit{binOpType}(\oplus, \tau_1^{\oplus}, \tau_2^{\oplus})$ está definido y para cualquier otro par $\tau_1, \tau_2 \in T$ tal que $\mathit{binOpType}(\oplus, \tau_1, \tau_2)$ está definido, $\tau_1 \leq \tau_1^{\oplus}$ y $\tau_2 \leq \tau_2^{\oplus}$.

Las propiedades 1. y 2. serán tomadas en cuenta en la sección de trabajo futuro porque se consideran útiles para la prueba de un hipotético teorema de *type safety*. Los tipos $\tau_i^{\oplus}$ de la propiedad 3 serán utilizados explícitamente en la traducción al lenguaje extendido con casts en el Capítulo 3. La definición explícita para cada símbolo de operador del lenguaje se encuentra en el Anexo B.5.

3.1.8.4. Chequeo de estructuras de datos

El chequeo de las tuplas se sigue del esquema de reglas T_TUPLE para $n \geq 0$ (y con la convención $\Gamma_1 \dagger \cdots \dagger \Gamma_0 = \emptyset$)

$$\frac{\Delta; \Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \cdots \qquad \Delta; \Gamma \vdash^t e_n : \tau_n \Rightarrow \Gamma_n}{\Delta; \Gamma \vdash^t \{e_1, \ldots, e_n\} : \{\tau_1, \ldots, \tau_n\} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (T\_TUPLE)}$$

Notar que el contexto de tipos exportado en T_TUPLE se construye a partir de las variables exportadas en cada coordenada, dando prioridad a las ocurrencias más a la derecha. Es posible ilustrar este último punto con un ejemplo:

```
x # tipa como integer
{x = 1.0, x, y = 2} # tipa como {float, integer, integer}
x # tipa como float
y # tipa como integer
```

En la segunda coordenada el tipo de `x` es `integer` ya que fue tipado bajo el contexto previo $\Gamma = [x \mapsto \texttt{integer}]$. El contexto final es el resultado de la combinación de los contextos exportados en las tres coordenadas: $[x \mapsto \texttt{float}], \emptyset, [y \mapsto \texttt{integer}]$, dando como resultado el nuevo contexto $[x \mapsto \texttt{float}, y \mapsto \texttt{integer}]$.

En el ejemplo anterior es posible observar la falla en que incurre el sistema de Cassola et al. [2022], que toma como base el contexto original para la consolidación del contexto exportado. En su sistema, la segunda coordenada `x` exportaría $[x \mapsto \texttt{integer}]$ y el contexto exportado resultante sería $[x \mapsto \texttt{integer}, y \mapsto \texttt{integer}]$, lo cual resulta inconsistente con el hecho que, luego de la evaluación, `x` vale `1.0`.

La regla T_MAP para el chequeo de los mapas es análoga a la de las tuplas:

$$\frac{\Delta; \Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \cdots \qquad \Delta; \Gamma \vdash^t e_n : \tau_n \Rightarrow \Gamma_n}{\Delta; \Gamma \vdash^t \texttt{\%\{}k_1 \texttt{ => } e_1 \texttt{,} \ldots \texttt{,} k_n \texttt{ => } e_n\texttt{\}} : \texttt{\%\{}k_1 \texttt{ => } \tau_1 \texttt{,} \ldots \texttt{,} k_n \texttt{ => } \tau_n\texttt{\}} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (T\_MAP)}$$

Es importante mencionar el hecho que, a pesar de que los tipos de mapas a menos de permutación son "equivalentes" respecto el juicio de tipado[4], esto no es el caso con las expresiones de mapas, ya que dos permutaciones podrían diferir en el contexto exportado. Por ejemplo:

`%{1 => x = :a,2 => x = :b}` exporta el contexto $[x \mapsto \texttt{:b}]$

mientras que

`%{1 => x = :b,2 => x = :a}` exporta el contexto $[x \mapsto \texttt{:a}]$

Finalmente, para el chequeo de listas se introducen dos reglas T_ELIST y T_CONS:

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

$$\frac{\Delta; \Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta; \Gamma \vdash^t e_2 : \tau_2 \Rightarrow \Gamma_2 \qquad [\tau_1] \vee \tau_2 = \sigma}{\Delta; \Gamma \vdash^t [e_1 \,|\, e_2] : \sigma \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (T\_CONS)}$$

Hay dos cambios para estas reglas respecto a las de Cassola et al. [2022]:

1. T_ELIST devuelve el nuevo tipo `[ ]` en vez de $[\tau]$.
2. T_CONS (allí T_LIST) devuelve el supremo entre la cabeza ($[\tau_1]$) y la cola (τ_2), sin asumir que estos coinciden (allí se usa τ para ambos).

A diferencia de otros cambio, éste no modifica sustancialmente al sistema y se debe atribuir al pasaje del modo declarativo al algorítmico. En el modo declarativo, el juicio $\Delta; \Gamma \vdash^t e : \tau \Rightarrow \Gamma'$ indica únicamente que τ es válido para e (pueden ser varios), mientras en el modo algorítmico τ debe ser necesariamente el tipo valido más pequeño (y por lo tanto único):

[4] De hecho, este enunciado se puede hacer formal, definiendo la equivalencia $\tau \sim \sigma$ al decir que:
$\vdash^t m[\tau] \iff \vdash^t m[\sigma]$ para todo módulo $m[]$ con un hueco para algún tipo en una de sus especificaciones

1. El tipo más pequeño para la lista vacía es `[ ]`.
2. El tipo más pequeño entre una lista que contenga un elemento de tipo τ_1 y una cola de tipo τ_2 es el supremo entre $[\tau_1]$ y τ_2 (si éste está definido).

La regla T_CONS muestra las bondades del sistema actual para con las listas heterogéneas: una lista heterogénea será validada siempre y cuando todos sus elementos compartan información útil; i.e., cuando el supremo entre sus tipos está definido. Por ejemplo:

$$\emptyset;\emptyset \vdash^t \texttt{[1.0|[1|[ ]]]} : \texttt{[number]} \Rightarrow \emptyset$$

es un juicio válido ya que le corresponde la derivación:

$$\dfrac{\dfrac{}{\emptyset;\emptyset \vdash^t \texttt{1.0} : \texttt{float} \Rightarrow \emptyset} \qquad \overset{D}{\emptyset;\emptyset \vdash^t \texttt{[1|[ ]]} : \texttt{[integer]} \Rightarrow \emptyset} \qquad \texttt{[float]} \vee \texttt{[integer]} = \texttt{[number]}}{\emptyset;\emptyset \vdash^t \texttt{[1.0|[1|[ ]]]} : \texttt{[number]} \Rightarrow \emptyset}$$

$$\dfrac{\dfrac{}{\emptyset;\emptyset \vdash^t \texttt{1} : \texttt{integer} \Rightarrow \emptyset} \qquad \dfrac{}{\emptyset;\emptyset \vdash^t \texttt{[ ]} : \texttt{[ ]} \Rightarrow \emptyset} \qquad \texttt{[integer]} \vee \texttt{[ ]} = \texttt{[integer]}}{\emptyset;\emptyset \vdash^t \texttt{[1|[ ]]} : \texttt{[integer]} \Rightarrow \emptyset}$$

Notar que en la regla T_CONS esta implícito el hecho de que el supremo entre $[\tau_1]$ y τ_2 está definido; de lo contrario no aplica. Por ejemplo, la lista `[1.0|[:one|[ ]]]` no es válida, ya que el supremo entre `[float]` y `[:one]` no está definido. Más aún, esto también permite descartar listas mal formadas, como por ejemplo `[1|2]` dado que `[integer]` $\vee$ `integer` no está definido.

3.1.8.5. Chequeo del *pattern matching*

La regla correspondiente al chequeo del *pattern matching* es:

$$\dfrac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \Gamma;\tau \vdash^{pt} p : \sigma \Rightarrow \Gamma''}{\Delta;\Gamma \vdash^t p \texttt{ = } e : \sigma \Rightarrow \Gamma' \dagger \Gamma''}\ \text{(T\_MATCH)}$$

donde se observa que el tipo τ se utiliza -junto al contexto original Γ- como *input* al chequeo del patrón p, Γ como contexto de variables externas y τ como el tipo candidato. El contexto de variables del patrón se inicializa vacío y su resultado (Γ'') se combina con el contexto Γ' para ser exportado.

Para ilustrar el funcionamiento de la regla T_MATCH, se proponen algunos ejemplos:

```
x # tipa como float
x = 1 # tipa como integer
x # tipa como integer
```

Para la segunda línea de la secuencia de arriba, la derivación correspondiente es:

$$\dfrac{\emptyset;[x \mapsto \texttt{float}] \vdash^t \texttt{1} : \texttt{integer} \Rightarrow \emptyset \qquad [x \mapsto \texttt{float}];\texttt{integer} \vdash^{pt} \texttt{x} : \texttt{integer} \Rightarrow [x \mapsto \texttt{integer}]}{\emptyset;[x \mapsto \texttt{float}] \vdash^t \texttt{x = 1} : \texttt{integer} \Rightarrow [x \mapsto \texttt{integer}]}$$

`x` inicialmente tiene tipo `float`; al realizar la asignación a `1` se ignora el tipo inicial y pasa a tener el tipo `integer`.

```
x # tipa como float
^x = 1 # error de tipado
```

Cambiando `x` por `^x` el juicio de chequeo del patrón:

$$[x \mapsto \texttt{float}]; \texttt{integer} \vdash^{pt} \texttt{\^{}x} : \_ \Rightarrow \_$$

deja de ser válido porque al estar "pinneada", la variable `x` pasa a estar ligada al contexto externo con el tipo `float`, el cual es incompatible con el tipo asignado (`integer`).

```
x # tipa como number
^x = 1 # tipa como integer
x # tipa como number
```

A diferencia del ejemplo anterior, aquí el chequeo sí es correcto porque el tipo del contexto externo (`number`) es compatible con el de la expresión (`integer`). Esto se ve reflejado en la derivación:

$$\frac{\emptyset; [x \mapsto \texttt{number}] \vdash^{t} 1 : \texttt{integer} \Rightarrow \emptyset \qquad [x \mapsto \texttt{number}]; \texttt{integer} \vdash^{pt} \texttt{\^{}x} : \texttt{integer} \Rightarrow \emptyset}{\emptyset; [x \mapsto \texttt{number}] \vdash^{t} \texttt{\^{}x = 1} : \texttt{integer} \Rightarrow \emptyset}$$

```
x # tipa como integer
y # tipa como float
{z, x} = {x, y = "a"} # tipa como {integer, string}
x # tipa como string
y # tipa como string
z # tipa como integer
```

En el ejemplo de arriba se muestra el resultado de un chequeo exitoso para tuplas. La última regla en la derivación correspondiente es:

$$\frac{\begin{array}{c}\emptyset; [x \mapsto \texttt{integer}, y \mapsto \texttt{float}] \vdash^{t} \texttt{\{x,y = "a"\}} : \texttt{\{integer,string\}} \Rightarrow [y \mapsto \texttt{string}] \\ [x \mapsto \texttt{integer}, y \mapsto \texttt{float}]; \texttt{\{integer,"a"\}} \vdash^{pt} \texttt{\{z,x\}} : \texttt{\{integer,string\}} \Rightarrow [x \mapsto \texttt{string}, z \mapsto \texttt{integer}]\end{array}}{\emptyset; [x \mapsto \texttt{integer}, y \mapsto \texttt{float}] \vdash^{t} \texttt{\{z,x\} = \{x,y = "a"\}} : \texttt{\{integer,string\}} \Rightarrow [x \mapsto \texttt{string}, y \mapsto \texttt{string}, z \mapsto \texttt{integer}]}$$

3.1.8.6. Chequeo de estructuras de control

La regla T_SEQ a continuación permite chequear una secuencia $e_1 ; e_2$ formada por dos expresiones:

$$\frac{\Delta; \Gamma \vdash^{t} e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta; \Gamma \dagger \Gamma_1 \vdash^{t} e_2 : \tau_2 \Rightarrow \Gamma_2}{\Delta; \Gamma \vdash^{t} e_1 ; e_2 : \tau_2 \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (T\_SEQ)}$$

En T_SEQ, el chequeo de la expresión en la izquierda ocurre primero y modifica, posiblemente, el contexto que su utiliza para chequear las segunda. Observar también que el chequeao de la segunda expresión determina el tipo resultante.

A diferencia de la regla T_ES de Cassola et al. [2022], el contexto para e_2 es $\Gamma \dagger \Gamma_1$ y no Γ_1 ya que aquí Γ_1 contiene únicamente las asignaciones explícitas en e_1 y para obtener el nuevo contexto se debe combinar con el Γ original. Por ese mismo motivo también sucede que el contexto exportado es $\Gamma_1 \dagger \Gamma_2$ en lugar de Γ_2.

Para una secuencia arbitraria e_1 ; ... ; e_n, T_SEQ se itera estructuralmente asociando a la derecha; i.e., aplicando la regla $n-1$ veces para e_1 ;$(\cdots(e_{n-1}$;$e_n)\cdots)$.

A continuación se expone un ejemplo de secuencia tipada:

```
x = 1
y = x + 1.0
x = y < x
```

Al ejemplo le corresponde la siguiente derivación:

$$\frac{D_0 \qquad \vdots \\ \emptyset;[x \mapsto \texttt{integer}, y \mapsto \texttt{float}] \vdash^t \texttt{x = y < x} : \texttt{boolean} \Rightarrow [x \mapsto \texttt{boolean}]}{\emptyset;\emptyset \vdash^t \texttt{x = 1; y = x + 1.0; x = y < x} : \texttt{boolean} \Rightarrow [x \mapsto \texttt{boolean}, y \mapsto \texttt{float}]}$$

donde D_0 es:

$$\frac{\vdots \qquad\qquad \vdots \\ \emptyset;\emptyset \vdash^t \texttt{x = 1} : \texttt{integer} \Rightarrow [x \mapsto \texttt{integer}] \qquad \emptyset;[x \mapsto \texttt{integer}] \vdash^t \texttt{y = x + 1.0} : \texttt{float} \Rightarrow [y \mapsto \texttt{float}]}{\emptyset;\emptyset \vdash^t \texttt{x = 1; y = x + 1.0} : \texttt{float} \Rightarrow [x \mapsto \texttt{integer}, y \mapsto \texttt{float}]}$$

Para el if/else, la regla de chequeo es básicamente la adaptación de TA-IF planteada en el Ejercicio 16.3.2 de Pierce [2002].

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \quad \tau \leq \texttt{boolean} \\ \Delta;\Gamma \dagger \Gamma' \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \quad \Delta;\Gamma \dagger \Gamma' \vdash^t e_2 : \tau_2 \Rightarrow \Gamma_2 \quad \tau_1 \vee \tau_2 = \sigma}{\Delta;\Gamma \vdash^t \texttt{if } e \texttt{ do } e_1 \texttt{ else } e_2 \texttt{ end} : \sigma \Rightarrow \Gamma'} \text{ (T\_IFE)}$$

El supremo $\tau_1 \vee \tau_2$ juega un papel similar al que ocurre en T_CONS, en cuanto permite conciliar tipos distintos para las ramas, siempre y cuando ambos compartan información útil; i.e., cuando el supremo σ está definido-.

En la misma línea, la hipótesis $\tau \leq$ `boolean` permite el chequeo exitoso en los casos particulares en que e_1 tiene tipo `:true` o `:false`, que es información suficiente para decidir una de las ramas.

La regla T_CASE para la definición por casos es:

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \\ \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_1 \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1' \quad \cdots \quad \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_n \vdash^t e_n : \tau_n \Rightarrow \Gamma_n' \\ \sigma = \bigvee_{i=1}^{n} \tau_i}{\Delta;\Gamma \vdash^t \texttt{case } e \texttt{ do } p_1 \texttt{ -> } e_1 \texttt{; ... ; } p_n \texttt{ -> } e_n \texttt{ end} : \sigma \Rightarrow \Gamma'} \text{ (T\_CASE)}$$

Aunque es muy similar a la de Cassola et al. [2022], la nueva T_CASE expresa su contenido computacional más directamente gracias al uso del supremo. Además, se debe agregar una capa extra de contextos en las hipótesis ya que el cambio en la semántica del contexto exportado obliga a tomar en cuenta el contexto original Γ en todas ellas.

3.1.8.7. Chequeo del operador de captura

El chequeo del operador de captura se basa en la regla T_ANON:

$$\frac{\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0)}{\Delta; \Gamma \vdash^t \&f\_name/n : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \emptyset} \text{ (T\_ANON)}$$

La regla T_ANON otorga el tipo de función a una captura según la especificación correspondiente en el contexto de funciones Δ. Para ser correcto el juicio, el par (f_name, n) debe existir como clave en Δ, aunque esto ya lo garantiza el compilador de Elixir -éste chequea que las capturas apunten a declaraciones de función existentes en el módulo en cuestión-.

3.1.8.8. Chequeo de llamadas a funciones

Para las llamadas a funciones declaradas en el módulo, el chequeo se basa en la regla D_CALL, para $n \geq 0$:

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \Delta; \Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Delta; \Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n \\ \sigma_1 \leq \tau_1 \quad \cdots \quad \sigma_n \leq \tau_n\end{array}}{\Delta; \Gamma \vdash^t f\_name(e_1, \ldots, e_n) : \tau_0 \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (T\_DCALL)}$$

Notar que la regla T_DCALL tiene $2n + 1$ puntos de falla:

1. Un punto de falla en el caso $(f\_name, n) \notin \Delta$
2. n puntos de falla para los chequeos recursivos $\Delta; \Gamma \vdash^t e_i : \sigma_i \Rightarrow \Gamma_i$
3. n puntos de falla en la comprobación del subtipado $\sigma_i \leq \tau_i$

Para funciones anónimas está la regla T_ACALL, que difiere con T_DCALL en que la información de tipos para la expresión aplicada proviene del chequeo de tipos de dicha expresión y no del contexto Δ:

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash^t e : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \Gamma_0 \\ \Delta; \Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Delta; \Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n \\ \sigma_1 \leq \tau_1 \quad \cdots \quad \sigma_n \leq \tau_n\end{array}}{\Delta; \Gamma \vdash^t e.(e_1, \ldots, e_n) : \tau_0 \Rightarrow \Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (T\_ACALL)}$$

Observar que T_ACALL tiene un punto de falla extra respecto a T_DCALL al afirmar que el tipo inferido para e es de la forma $(\tau_1, .., \tau_n) \to \tau_0$, pues podría ser el caso que e se chequea exitosamente pero:

1. O bien no es de tipo función.
2. O bien es de la forma $(\tau_1, .., \tau_m) \to \tau_0$, com $m \neq n$.

El item 2. no aplica de ningú modo a T_DCALL, porque por construcción de Δ en j. colección de especificaciones en declaración, el n que refiere a $(\tau_1, ..., \tau_n)$ debe coincidir con el n de (f_name, n).

3.1.9. Reglas para el chequeo-refinamiento de patrones

A diferencia de las reglas para el chequeo de expresiones, las reglas para el chequeo de patrones respecto al juicio j. chequeo-refinamiento de patrones no buscan deshacerse de todas las evaluaciones erroneas. Las condiciones que determinan un `MatchError` están íntimamente ligadas a la casuística del *runtime*, de forma tal que su prevención completa resiste cualquier análisis estático razonable. Por ejemplo, suponer el siguiente escenario:

```
x # declarado en el contexto con tipo tau
1 = x
```

Notar que independiente del valor concreto del tipo τ, en ningún caso es posible deducir de $\Gamma[x] = \tau$ el resultado del *matching* `1 = x`.

Por lo tanto, las reglas asumen una política de *best-effort* respecto a la prevención de errores en la evaluación, rechazando los casos que sean posibles a partir de la información de tipos y el patrón en cuestión. Para el ejemplo anterior, la decisión que se toma es la de es exigir $\tau \geq$ `integer`, lo que partiría el escenario anterior en dos:

Escenario 1: $\tau \geq$ `integer`:

```
x # declarado en el contexto con tipo integer o number
1 = x # exito del chequeo estático del patrón
```

El chequeo es exitoso porque el *matching* `1 = x` tiene posibilidades de ser exitoso en tiempo de ejecución (si `x` vale 1).

Escenario 2: $\tau \not\geq$ `integer`:

```
x # declarado en el contexto con tipo distinto de integer y number
1 = x # falla del chequeo estático del patrón
```

El chequeo falla porque la información estática disponible es suficiente para determinar que el *matching* `1 = x` lanzará un `MatchError` en tiempo de ejecución.

De los dos casos, se concluye que la condición $\tau \geq$ `integer` captura todos los escenarios estáticos donde el *matching* es viable de ser exitoso en tiempo de ejecución.

Para el juicio de chequeo-refinamiento hay una única regla TCP, que se formula como

$$\frac{\Sigma; \emptyset; \tau \vdash^{pc} p \Rightarrow \Gamma \qquad \Sigma; \Gamma; \tau \vdash^{ptv} p : \sigma}{\Sigma; \tau \vdash^{pt} p : \sigma \Rightarrow \Gamma} \text{ (TCP)}$$

Ésta indica que primero se deben recolectar exitosamente las variables en p respecto al tipo candidato τ en un contexto Γ, y luego usar ese mismo contexto para inferir el tipo σ usando p como "guía". En todo momento es importante, además, considerar el contexto externo Σ para las variables "pinneadas". Aunque el formato del juicio no se inspira en otros ejemplos de la literatura, en el Anexo B.10 se explora un vínculo indirecto con la disciplina de *Ocurrence Typing* desarrollada en Tobin-Hochstadt and Felleisen [2010].

3.1.9.1. Reglas para la colección

El juicio j. colección de variables en patrones es la versión estática del proceso de asignación de variables dentro del patrón durante la evaluación. A la vez de recolectar los tipos para las variables de p a partir de τ, se debe verificar en todo momento que τ tenga sentido para la forma de p en cuestión. Las reglas base son:

$$\frac{type(l) \leq \tau}{\Sigma; \Gamma; \tau \vdash^{pc} l \Rightarrow \Gamma} \text{(TPC\_LIT)} \qquad \frac{\Sigma[x] = \sigma \quad \tau \wedge \sigma = \mu}{\Sigma; \Gamma; \tau \vdash^{pc} \hat{}x \Rightarrow \Gamma} \text{(TPC\_PIN)} \qquad \frac{}{\Sigma; \Gamma; \tau \vdash^{pc} \_ \Rightarrow \Gamma} \text{(TPC\_WILD)}$$

$$\frac{x \notin \Gamma}{\Sigma; \Gamma; \tau \vdash^{pc} x \Rightarrow \Gamma[x \mapsto \tau]} \text{(TPC\_VARE)} \qquad \frac{\Gamma[x] = \sigma \quad \tau \wedge \sigma = \mu}{\Sigma; \Gamma; \tau \vdash^{pc} x \Rightarrow \Gamma[x \mapsto \mu]} \text{(TPC\_VARN)}$$

Cuando el patrón p es un literal l, no hay colección posible y TPC_LIT únicamente chequea que el tipo candidato sea un supertipo del tipo $type(l)$ del literal

TPC_PIN es similar a TPC_LIT en cuanto no hay colección posible y el chequeo de `^x` involucra una restricción para el tipo candidato. En vez de tomar $\sigma \leq \tau$ para la restricción (siendo $\Sigma[x] = \sigma$), se exige que $\tau \wedge \sigma$ esté definido, que se traduce a tener *compatibilidad* entre el tipo externo σ y el candidato τ.[5].

Cuando el patrón es _ no hay colección ni chequeo posible, por lo que TPC_WILD constituye una regla trivial.

La regla TPC_VARE recolecta en Γ el tipo candidato para la variable dada por el patrón x en el caso que ésta no haya sido previamente recolectada ($x \notin \Gamma$). En caso que la variable x ya fuera recolectada, TPC_VARN chequea además que el nuevo tipo candidato τ sea compatible con el previamente existente (σ). En ese caso $\Gamma[x]$ se refina al tipo común entre ambos ($\tau \wedge \sigma$).

En el sistema de tipos actual, el interés del refinamiento usando el operador de ínfimo $\wedge$ puede verse en la práctica; por ejemplo si se utilizan mapas para modelar registros de usuarios. En caso que el tipo candidato para `x` sea `%{:age => integer}` y el previamente recolectado sea `%{:name => string}`, la regla anterior refina el tipo de `x` en `%{:age => integer,:name => string}`.

Para los patrones de listas hay dos reglas, una por cada constructor:

$$\frac{\Sigma; \Gamma; \tau \vdash^{pc} p_1 \Rightarrow \Gamma' \quad \Sigma; \Gamma'; [\tau] \vdash^{pc} p_2 \Rightarrow \Gamma''}{\Sigma; \Gamma; [\tau] \vdash^{pc} [p_1 \mid p_2] \Rightarrow \Gamma''} \text{(TPC\_CONS)} \qquad \frac{[\,] \leq \tau}{\Sigma; \Gamma; \tau \vdash^{pc} [\,] \Rightarrow \Gamma} \text{(TPC\_ELIST)}$$

TPC_CONS aplica cuando el tipo candidato tiene forma de lista no vacía $[\tau]$; en ese caso se ejecuta el chequeo del patron de la cabeza con τ y el de la cola con $[\tau]$, recolectando las variables que ocurran en uno y luego en otro de manera secuencial.

TPC_ELIST es análogo a TPC_LIT, tomando en cuenta que `[ ]` es el menor tipo posible para la lista vacía `[ ]` (ahora en el rol de patrón).

Para las tuplas, existe una única regla que aplica para cada $n \geq 0$:

$$\frac{\Sigma; \Gamma; \tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Sigma; \Gamma_{n-1}; \tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n}{\Sigma; \Gamma; \{\tau_1, \ldots, \tau_n\} \vdash^{pc} \{p_1, \ldots, p_n\} \Rightarrow \Gamma_n} \text{(TPC\_TUPLE)}$$

La regla TPC_TUPLE chequea primero que el tipo candidato sea una tupla de tamaño n (el de p) para luego pasar a aplicar recursivamente las reglas del sistema iterando sobre todas las componentes p_i y su respectiva coordenada de tipo candidata τ_i.

[5]Se podría argumentar que la misma exigencia aplica en TPC_LIT y chequear en vez que $\tau \wedge type(l)$ esté definido, pero como $type(l)$ siempre es minimal basta con chequear $type(l) \leq \tau$, que es computacionalmente más sencillo

A modo de ejemplo, se describe a continuación la derivación que corresponde a

$$\emptyset;\emptyset;\{\texttt{number},\texttt{integer},\texttt{atom}\} \vdash^{pc} \{\texttt{x},\texttt{x},\texttt{y}\} \Rightarrow [x \mapsto \texttt{integer}, y \mapsto \texttt{atom}]$$

La última regla aplicada es una instancia de TPC_TUPLE con $n = 3$:

$$\frac{\begin{array}{c}\emptyset;\emptyset;\texttt{number} \vdash^{pc} \texttt{x} \Rightarrow [x \mapsto \texttt{number}] \\ \emptyset;[x \mapsto \texttt{number}];\texttt{integer} \vdash^{pc} \texttt{x} \Rightarrow [x \mapsto \texttt{integer}] \\ \emptyset;[x \mapsto \texttt{integer}];\texttt{atom} \vdash^{pc} \texttt{y} \Rightarrow [x \mapsto \texttt{integer}, y \mapsto \texttt{atom}]\end{array}}{\emptyset;\emptyset;\{\texttt{number},\texttt{integer},\texttt{atom}\} \vdash^{pc} \{\texttt{x},\texttt{x},\texttt{y}\} \Rightarrow [x \mapsto \texttt{integer}, y \mapsto \texttt{atom}]}$$

donde la primera y la tercera hipótesis se verifican directamente por TPC_VARE y la segunda por una instancia de TPC_VARN:

$$\frac{[x \mapsto \texttt{number}][x] = \texttt{number} \quad \texttt{integer} \wedge \texttt{number} = \texttt{integer}}{\emptyset;[x \mapsto \texttt{number}];\texttt{integer} \vdash^{pc} \texttt{x} \Rightarrow [x \mapsto \texttt{integer}]}$$

Del ejemplo anterior se observa, en contraste, que

$$\emptyset;\emptyset;\{\texttt{integer},\texttt{float},\texttt{atom}\} \vdash^{pc} \{\texttt{x},\texttt{x},\texttt{y}\} \Rightarrow \_$$

no se verifica ya que `integer` y `float` son tipos incompatibles para `x`. En efecto, en TPC_VARE el ínfimo `integer` $\wedge$ `float` no está definido.

La regla para mapas aplica para $n \geq 0$, con la convención previa de que $\{1, ..., 0\}$ denota al conjunto vacío.

$$\frac{\begin{array}{c}(k_{i_1}, .., k_{i_n}) = (l_1, .., l_n) \text{ con } \{i_1, .., i_n\} \subseteq I \\ \Sigma;\Gamma;\tau_{k_{i_1}} \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Sigma;\Gamma_{n-1};\tau_{k_{i_n}} \vdash^{pc} p_n \Rightarrow \Gamma_n\end{array}}{\Sigma;\Gamma;\,\%\{\overline{(k_i \texttt{ => } \tau_i)}_{i\in I}\} \vdash^{pc} \%\{\overline{(l_i \texttt{ => } p_i)}_{i\in\{1,..,n\}}\} \Rightarrow \Gamma_n} \text{ (TPC\_MAP)}$$

A diferencia de Cassola et al. [2022], donde TPC_MAP es completamente análoga a TPC_TUPLE, aquí la regla debe modificarse porque el sistema actual es algorítmico y como tal, prescinde de una regla de subsumption -llamada T_SUB en ese trabajo- que permita validar todos los casos deseados. La modificación de TPC_MAP consiste en relajar la hipótesis de conjuntos de claves idénticos a pedir únicamente que las claves de p sean un subconjunto de las de τ. De allí el chequeo prosigue análoga a TPC_TUPLE usando las claves $(k_i)_{i\in I}$ en el rol de las coordenadas.

A continuación se expone un ejemplo que refleja los cambios en la regla TPC_MAP:

$$\frac{\dfrac{\texttt{:b} \leq \texttt{atom}}{[x \mapsto \texttt{atom}];\emptyset;\texttt{atom} \vdash^{pc} \texttt{:b} \Rightarrow \emptyset}\text{ (TPC\_LIT)} \qquad \dfrac{\texttt{:a} \wedge \texttt{atom} = \texttt{:a}}{[x \mapsto \texttt{atom}];\emptyset;\texttt{:a} \vdash^{pc} \texttt{\^{}x} \Rightarrow \emptyset}\text{ (TPC\_PIN)}}{[x \mapsto \texttt{atom}];\emptyset;\,\%\{1 \texttt{ => } \texttt{:a},2 \texttt{ => } \texttt{atom},3 \texttt{ => } \texttt{:c}\} \vdash^{pc} \%\{2 \texttt{ => } \texttt{:b},1 \texttt{ => } \texttt{\^{}x}\} \Rightarrow \emptyset}\text{ (TPC\_MAP)}$$

En Cassola et al. [2022], el juicio se hubiera verificado con τ = `%{2 => atom,1 => :a}`, y se precisaría además la regla de subsumption para validar el *pattern matching* a partir de una expresión e de tipo `%{1 => :a,2 => atom,3 => :c}`.

3.1.9.2. Reglas para el refinamiento

Una vez construido el contexto de variables siguiendo las reglas del juicio de colección, el juicio j. colección de variables en patrones construye el tipo σ combinando la información original del tipo candidato τ y el patrón p con las variables recolectadas en ese contexto. La reglas del Recuadro 3.2 muestran como la construcción de σ parte de τ, reemplazando cada subestructura τ_0 en los siguientes casos de la derivación de $\Sigma; \Gamma; \tau \vdash^{pc} p \Rightarrow \Gamma'$:

1. $\Gamma[x]$ cada vez que τ_0 fue candidato en la regla TPC_VARE o TPC_VARN (ver TPV_VARE y TPV_VARN)
2. $type(l)$ cada vez que τ_0 fue candidato en la regla TPC_LIT (ver TPV_LIT)
3. $\tau_0 \wedge \sigma$ cada vez que τ_0 fue candidato en la regla TPC_PIN (ver TPV_PIN)
4. `[ ]` cada vez que τ_0 fue candidato en la regla TPC_ELIST (ver TPV_ELIST)

Finalmente, en TPV_CONS se reemplaza el tipo $[\tau]$ cada vez que se aplicó TPC_CONS respecto al patrón $[p_1 \mid p_2]$:

5. $\sigma_1 \vee \sigma_2$ cada vez que τ se reemplazó por σ_1 respecto a p_1 y $[\tau]$ se reemplazó por σ_2 respecto p_2.

Para ilustrar el refinamiento, conviene introducir algunos ejemplos:

Ejemplo 1: De la derivación para

$$\emptyset; \emptyset; \{\texttt{number,integer,atom}\} \vdash^{pc} \{\texttt{x,x,y}\} \Rightarrow [x \mapsto \texttt{integer}, y \mapsto \texttt{atom}]$$

se obtiene el refinamiento

$$\emptyset; [x \mapsto \texttt{integer}, y \mapsto \texttt{atom}]; \{\texttt{number,integer,atom}\} \vdash^{ptv} \{\texttt{x,x,y}\} : \{\texttt{integer,integer,atom}\}$$

Ejemplo 2: De la derivación para

$$[x \mapsto \texttt{integer}]; \emptyset; \{\texttt{number,number,atom}\} \vdash^{pc} \{\texttt{x,\^{}x,y}\} \Rightarrow [x \mapsto \texttt{number}, y \mapsto \texttt{atom}]$$

se obtiene el refinamiento:

$$[x \mapsto \texttt{integer}]; [x \mapsto \texttt{number}, y \mapsto \texttt{atom}]; \{\texttt{number,number,atom}\} \vdash^{ptv} \{\texttt{x,\^{}x,y}\} : \{\texttt{number,integer,atom}\}$$

Ejemplo 3: De la derivación para

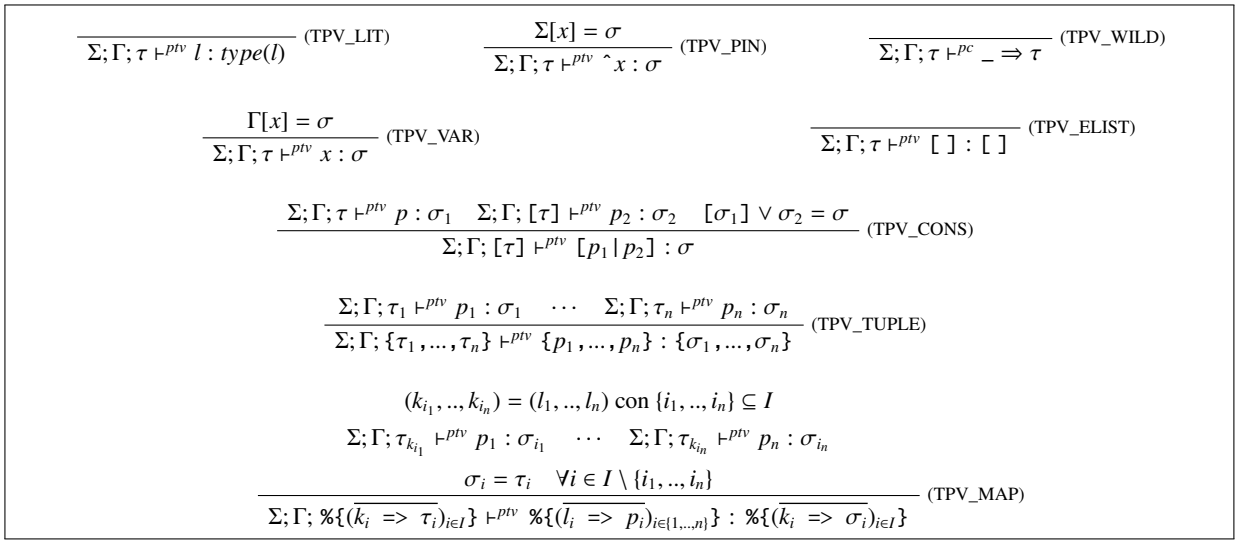
$$\emptyset; \emptyset; \texttt{\%\{:a => [atom],:b => number\}} \vdash^{pc} \texttt{\%\{:a => [ ]\}} \Rightarrow \emptyset$$

se obtiene el refinamiento:

$$\emptyset; \emptyset; \texttt{\%\{:a => [atom],:b => number\}} \vdash^{ptv} \texttt{\%\{:b => [ ]\}} : \texttt{\%\{:a => [ ],:b => number\}}$$

En general, se puede demostrar que cada vez que $\Sigma; \Gamma; \tau \vdash^{pc} p \Rightarrow \Gamma'$ es válido, $\Sigma; \Gamma; \tau \vdash^{ptv} p : \sigma$ también lo es (para cierto σ). De hecho, la derivación del juicio de refinamiento no contiene más[6] que una construcción directa de σ a partir de Σ, Γ', τ y p. En el Anexo B.9 se detalla el algoritmo explícito $\textsc{Refine}(\Sigma, \Gamma', \tau, p)$ que implementa esta construcción.

[6] En realidad, habría que probar que el supremo en TPV_LIST siempre está definido. Esto no es muy difícil y se posterga la prueba formal para un trabajo futuro



Recuadro 3.2: Reglas para el juicio de chequeo de patrones

3.1.10. Propiedades del chequeo-refinamiento de patrones

Primero que nada, es importante contar con una propiedad formal que justifique la alusión al refinamiento entre los tipos τ y σ en el j. chequeo-refinamiento de patrones:

Refinamiento de tipo en el chequeo de patrones: Cada vez que el juicio $\Sigma; \tau \vdash^{pt} p : \sigma \Rightarrow \_$ es válido, se cumple $\sigma \leq \tau$. Si además p no tiene variables repetidas, se cumple $\sigma = \tau$.

Bosquejo de la prueba: Es claro que luego de obtener $\Sigma; \emptyset; \tau \vdash^{pc} p \Rightarrow \Gamma$, Γ cumple $\Gamma[x] \leq \tau_0$, para cada $x \in \text{dom}(\Gamma)$ y cada subestructura τ_0 de τ donde se obtuvo $\Sigma; \_; x \vdash^{pc} \tau_0 \Rightarrow \_$. Luego, se prueba por inducción en $\Sigma; \Gamma; \tau \vdash^{ptv} p : \sigma$ que $\sigma \leq \tau$ usando la observación anterior para los casos base.

Además, para un trabajo futuro, resulta de interes probar que el chequeo-refinamiento de patrones goza de dos propiedades extra:

***Type safety* del chequeo de patrones:** Cada vez que el juicio $\Gamma; \tau \vdash^{pt} p : \sigma \Rightarrow \_$ es válido y v es un valor tal que tanto $\Delta; \emptyset \vdash^{t} v : \mu \Rightarrow \_$ como $\mu \leq \tau$ son correctos y la evaluación $match(\Gamma^*, \emptyset, p, v)$ es exitosa -siendo Γ^* un contexto de valores adecuado para Γ[7]-, entonces $\mu \leq \sigma$.

Usando la notación $V^\tau \equiv \{v \in V : \Delta; \emptyset \vdash^{t} v : \mu \Rightarrow \_ \ \wedge \ \mu \leq \tau\}$ para representar a los valores válidos para cierto tipo τ y $V_p \equiv \{v \in V : match(\Gamma^*, \emptyset, p, v)\}$ para los valores que se "matchean" exitosamente respecto al patrón p, la propiedad anterior se puede expresar en términos conjuntistas como:

$$\Gamma; \tau \vdash^{pt} p : \sigma \Rightarrow \_ \text{ implica } V^\tau \cap V_p \subseteq V^\sigma$$

[7] En el sentido de que $\text{dom}(\Gamma^*) \subseteq \text{dom}(\Gamma)$ y $\forall x \in \text{dom}(\Gamma^*)\ \Delta; \emptyset \vdash^{t} \Gamma^*[x] : \mu \leq \Gamma[x] \Rightarrow \_$

No vacuidad del chequeo de patrones respecto al *runtime*: Cada vez que el juicio $\Gamma; \tau \vdash^{pt} p : \sigma \Rightarrow \_$ es válido y Γ^* es un contexto de valores adecuado para Γ, existe un valor v -que depende de Δ- tal que $\Delta; \emptyset \vdash^{t} v : \sigma \Rightarrow \_$ es válido y la evaluación $match(\Gamma^*, \emptyset, p, v)$ es exitosa.

De lograr ser probada, y en vista de la propiedad anterior, la no vacuidad constituye una propiedad de optimalidad respecto al tipo σ encontrado por el refinamiento. A continuación se intenta hacer precisa esta idea:

Se podría pensar en un "refinamiento dinámico" dados τ y p definido por

$$refine(\Gamma^*, \tau, p) \equiv \min\{\mu \in T : V^\tau \cap V_p \subseteq V^\mu\}^{8}$$

esto es, $refine(\Gamma^*, \tau, p)$ constituye el menor tipo que captura desde el sistema estático a los valores de τ que se "matchean" exitosamente respecto a p. En ese caso, la propiedad de *type safety* se puede reformular como[9]

$$\Gamma; \tau \vdash^{pt} p : \sigma \Rightarrow \_ \text{ y } \Gamma^* \text{adecuado para } \Gamma \text{ implica } refine(\Gamma^*, \tau, p) \leq \sigma$$

Por otra parte, la propiedad de no vacuidad encuentra un valor v de tipo *exactamente* σ, luego se puede afirmar *a posteriori* que:

$$\Gamma; \tau \vdash^{pt} p : \sigma \Rightarrow \_ \text{ y } \Gamma^* \text{adecuado } \Gamma \text{ implica } refine(\Gamma^*, \tau, p) = \sigma$$

La formulación de las dos propiedades anteriores depende de la definición de la sintaxis para los valores v y de la definición de la evaluación de patrones *match*, ambas planteadas en la Sección 4.2.

3.2. Extensión Gradual

Esta sección será dedicada a la formulación de un nuevo sistema, desarrollado a partir del anterior, proponiendo cambios para incorporar a su diseño los principios del *gradual typing*.

Presentado en Siek and Taha [2006] como una alternativa para compatiblizar los chequeos de tipado con la presencia de porciones de código dinámicas, los sistemas de tipos basados en *gradual typing* integran características del tipado dinámico añadiendo a la sintaxis de tipos del lenguaje el símbolo "?" para representar al *tipo dinámico*, que, como tal, contiene información cuya consideración debe ser delegada al *runtime*.

Esencialmente, un sistema de tipos gradual (es decir, basado en *gradual typing*) no difiere demasiado de un sistema de tipos estático más que en tener en cuenta los siguientes aspectos:

1. El tipo dinámico es un "ciudadano de primer orden" de la sintaxis de tipos; esto significa que puede ser combinado con otros tipos y constructores para representar información parcialmente dinámica:

   `[?]` `{?,integer}` `{?,?,?}` `? → atom` `%{1 => ?}`

   serían ejemplos de tipos parcialmente dinámicos dentro de la sintaxis de este trabajo.

[8] Como $\tau \mapsto V^\tau$ es el adjunto superior de una Conexión de Galois, la definición siempre tiene sentido

[9] Por definición de V^τ, es claro que el mapeo $\tau \mapsto V^\tau$ refleja desigualdades

2. La relación de *consistencia* $\tau \sim \sigma$ entre tipos graduales como reemplazo de la igualdad sintáctica: En presencia de información dinámica, los análisis estáticos dependerán de identificar pares de tipos que son *plausibles de igualdad* en tiempo de ejecución:

$$\texttt{?} \sim \texttt{integer} \quad \texttt{float} \sim \texttt{?} \quad \texttt{\{?,integer\}} \sim \texttt{\{integer,?\}}$$

   serían ejemplos de pares de consistencia utilizando tipos dentro de la sintaxis de este trabajo.

3. Nuevas reglas (o cambios en las existentes) que permiten validar implícitamente juicios de chequeo de expresiones a menos de la consistencia; i.e.:

$$\frac{\Delta; \Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \tau \sim \sigma}{\Delta; \Gamma \vdash^t e : \sigma \Rightarrow \Gamma'}$$

En vez de "?", la gradualización del sistema de la sección anterior utiliza `any` para el tipo dinámico, ya que el primero no representa un símbolo de tipo válido dentro de Elixir. La relación de consistencia $\sim$ tiene una definición estándar respecto a la literatura y, aunque ésta no figura explícitamente en las nuevas reglas, constituye la base para la definición de nuevos "predicados y funciones consistentes" que sustituyen a los predicados auxiliares al sistema estático (subtipado, supremo, etc.). Esta sustitución consistuye el principal cambios, aunque son necesarios algunos otros para lograr un sistema equiparable al de Cassola et al. [2022]:

- Para el juicio de patrones se agrega la nueva regla TPC_GROUND, que se hace necesaria en sustitución de la regla de subsumption gradual T_DOWN de Cassola et al. [2022] con el fin de mantener un sistema algorítmico.
- Deben introducirse nuevas reglas para soportar módulos *parcialmente especificados*; i.e., con especificaciones faltantes para algunas de las declaraciones de función.

A posteriori, el nombre de la sección se justifica porque, como se verá, las modificaciones son conservativas respecto al sistema estático de base: cada programa m chequeado exitosamente respecto al sistema estático también lo será respecto al sistema gradual.

3.2.1. Extensión de la sintaxis de tipos

Como se mencionó anteriormente, la nueva sintaxis de tipos ($\tilde{T}$) se mantiene sin modificaciones respecto a la versión estática (T), con la única diferencia que se incluye un nuevo caso base para `any`. Mientras que los documentos oficiales promueven el uso de `any` -aunque de forma un poco vaga - como un tipo máximo, aquí se escoge para ocupar el rol del tipo dinámico, elemento distintivo de la disciplina del *gradual typing*.

$$\begin{array}{lcl} \tilde{T} & ::= & B \mid \texttt{any} \mid \texttt{[ ]} \mid \texttt{[}\tilde{T}\texttt{]} \\ & \mid & \texttt{\{}\tilde{T}\texttt{,...,}\tilde{T}\texttt{\}} \mid \texttt{\%\{}l\texttt{ => }\tilde{T}\texttt{,...,}l\texttt{ => }\tilde{T}\texttt{\}} \mid (\tilde{T}, .., \tilde{T}) \to \tilde{T} \\ G & ::= & B \mid \texttt{any} \mid \texttt{[ ]} \mid \texttt{[any]} \\ & \mid & \texttt{\{any,...,any\}} \mid \texttt{\%\{}l\texttt{ => any,...,}l\texttt{ => any\}} \mid (\texttt{any}, .., \texttt{any}) \to \texttt{any} \end{array}$$

En G se representa la subsintaxis de $\tilde{T}$ para los tipos conocidos como *ground* en varios artículos de la literatura (por ejemplo, en la Sección 2.2.3 de Siek et al. [2015]). Estos serán importantes para la definición del cast en el capítulo siguiente y además jugarán un rol en la definición de la regla TPC_GROUND.

3.2.2. Relaciones del sistema

Para construir la extensión gradual del sistema de tipos, será necesario hacer mención a dos relaciones básicas entre tipos graduales bien conocidos en la literatura: la de *consistencia* ($\sim$) y la de *precisión* ($\lll$). Además, ambas serán combinadas con el subtipado para formar las relaciones de *subtipado consistente* ($\lesssim$) -también de la literatura- y *subtipado precisión* ($<^{-}$).

3.2.2.1. Relación de consistencia

La relación de consistencia está definida para cada par de tipos graduales $\tilde{T}$. La definición es estándar con respecto a la literatura (por ejemplo, Garcia et al. [2016]) y debe pensarse como una "igualdad relajada" entre tipos en presencia de información dinámica.

$$\frac{}{\texttt{any} \sim \tau}\ \text{(C\_ANY\_L)} \qquad \frac{}{\tau \sim \texttt{any}}\ \text{(C\_ANY\_R)} \qquad \frac{}{\tau \sim \tau}\ \text{(C\_REFL)}$$

$$\frac{\tau \sim \sigma}{[\tau] \sim [\sigma]}\ \text{(C\_LIST)} \qquad \frac{\tau_i \sim \sigma_i \quad \forall i \in \{1,...,n\}}{\{\tau_1,...,\tau_n\} \sim \{\sigma_1,...,\sigma_n\}}\ \text{(C\_TUPLE)}$$

$$\frac{\tau_i \sim \sigma_i \quad \forall i \in \{1,...,n\}}{\texttt{\%\{}(\overline{k_i \texttt{ => } \tau_i})_{i\in I}\texttt{\}} \sim \texttt{\%\{}(\overline{k_i \texttt{ => } \sigma_i})_{i\in I}\texttt{\}}}\ \text{(C\_MAP)} \qquad \frac{\tau_i \sim \sigma_i \quad \forall i \in \{0,...,n\}}{(\tau_1,..,\tau_n) \to \tau_0 \sim (\sigma_1,..,\sigma_n) \to \sigma_0}\ \text{(C\_FUN)}$$

Informalmente, puede afirmarse que dos tipos son consistentes si únicamente difieren en información dinámica. A continuación algunos ejemplos:

$$\texttt{\{any,integer\}} \sim \texttt{\{atom,any\}},\ \texttt{[any]} \sim \texttt{[[any]]},\ \texttt{any} \to \texttt{integer} \sim \texttt{any} \to \texttt{any}$$

$$\texttt{\{integer,any\}} \not\sim \texttt{\{atom,any\}},\ \texttt{[any]} \not\sim \texttt{\{any\}},\ \texttt{any} \to \texttt{integer} \not\sim () \to \texttt{any}$$

Observar además que a diferencia de las reglas presentadas anteriormente para los mapas; la consistencia sí requiere el mismo orden de las claves. Por ejemplo, se tiene

$$\texttt{\%\{1 => integer,2 => any\}} \sim \texttt{\%\{1 => any,2 => atom\}}$$

mientras que

$$\texttt{\%\{2 => any,1 => integer\}} \not\sim \texttt{\%\{1 => any,2 => atom\}}$$

$$\texttt{\%\{1 => integer\}} \not\sim \texttt{\%\{1 => any,2 => atom\}}$$

Aunque anteriormente se ha dado la idea de que la consistencia funciona como una igualdad relajada, debe advertirse que esto no es algo preciso y debe tomarse únicamente a modo de intuición. En términos formales, la consistencia no es una relación de equivalencia ya que no verifica la propiedad transitiva. Por ejemplo, `integer` $\sim$ `any` y `any` $\sim$ `atom` mientras que `integer` $\not\sim$ `atom`.

3.2.2.2. Relación de precisión

La relación de consistencia es muy cercana a la relación de *precisión* $\tau \lll \sigma$ que se introduce en Siek et al. [2015] (con el símbolo $\sqsubseteq$) y luego reintroducida en Cassola et al. [2022] con la notación actual. Las reglas son exactamente iguales a las de $\tau \sim \sigma$ con la diferencia de que se remueve el caso base C_ANY_L, con lo que $\lll$ deja de ser simétrica y se convierte en una relación de orden. Intuitivamente $\tau \lll \sigma$ si σ es el resultado de "olvidar" cualquier cantidad de información estática de τ en favor del tipo dinámico.

$$\frac{}{\tau \lll \texttt{any}}\ \text{(P\_ANY)} \qquad \frac{}{\tau \lll \tau}\ \text{(P\_REFL)}$$

$$\frac{\tau \lll \sigma}{[\tau] \lll [\sigma]}\ \text{(P\_LIST)} \qquad \frac{\tau_i \lll \sigma_i \quad \forall i \in \{1,...,n\}}{\{\tau_1,...,\tau_n\} \lll \{\sigma_1,...,\sigma_n\}}\ \text{(P\_TUPLE)}$$

$$\frac{\tau_i \lll \sigma_i \quad \forall i \in \{1,...,n\}}{\%\{(\overline{k_i \texttt{ => } \tau_i})_{i\in I}\} \lll \%\{(\overline{k_i \texttt{ => } \sigma_i})_{i\in I}\}}\ \text{(P\_MAP)} \qquad \frac{\tau_i \lll \sigma_i \quad \forall i \in \{0,...,n\}}{(\tau_1,..,\tau_n) \to \tau_0 \lll (\sigma_1,..,\sigma_n) \to \sigma_0}\ \text{(P\_FUN)}$$

En Cassola et al. [2022], la relación de precisión juega un papel central ya que constituye la base de la definición de la regla de subsumption gradual T_DOWN. En éste trabajo, la relación de precisión es necesaria para poder expresar el criterio SGG en la Sección 3.2.8.

Como se verá más adelante, el criterio SGG se formula al nivel de módulo, por lo que es necesario extender la precisión a dicho nivel. Esto se hace siguiendo la misma filosofía que Siek et al. [2015], pero adaptada al contexto de este trabajo:

$$\frac{\begin{array}{c}\{1,...,n\} \subseteq \{i_1,...,i_m\} \quad d_1 \lll d'_1 \quad \cdots \quad d_n \lll d'_n \\ \{d_{i_1},...,d_{i_m}\} \setminus \{d_1,...,d_n\} = \{s_1,...,s_k\} \\ m = \texttt{defmodule Program do } d_{i_1}\texttt{, ..., } d_{i_m} \texttt{ end} \\ m' = \texttt{defmodule Program do } d'_1\texttt{, ..., } d'_n \texttt{ end}\end{array}}{m \lll m'}\ \text{(P\_PROG)}$$

Notar que P_PROG establece que un módulo m puede perder precisión omitiendo algunas especificaciones $s_1,...,s_k$, siempre que se siga respetando la precisión $d_i \lll d'_i$ para las que sobreviven. Las reglas que definen la precisión para declaraciones son:

$$\frac{}{\texttt{def } f\_name(p_1,...,p_n) \texttt{ do } e \texttt{ end} \lll \texttt{def } f\_name(p_1,...,p_n) \texttt{ do } e \texttt{ end}}\ \text{(P\_DEF)}$$

$$\frac{\tau_i \lll \sigma_i \quad \forall i \in \{0,...,n\}}{\texttt{@spec } f\_name(\tau_1,...,\tau_n) \texttt{ :: } \tau_0 \lll \texttt{@spec } f\_name(\sigma_1,...,\sigma_n) \texttt{ :: } \sigma_0}\ \text{(P\_SPEC)}$$

Eventualmente, se precisará extender la precisión a contextos de dos maneras sutilmente distintas:

$$\Gamma \lll \Gamma' ::= \text{dom}(\Gamma) = \text{dom}(\Gamma') \land \forall x \in \text{dom}(\Gamma), \Gamma[x] \lll \Gamma'[x] \qquad \text{(precisión de contextos)}$$

$$\Gamma \lll\!\!\!\lll \Gamma' ::= \text{dom}(\Gamma) \supseteq \text{dom}(\Gamma') \land \forall x \in \text{dom}(\Gamma'), \Gamma[x] \lll \Gamma'[x] \qquad \text{(superprecisión de contextos)}$$

3.2.2.3. Relación de subtipado consistente

Para extender el subtipado a los tipos graduales, se añade un único axioma

$$\frac{}{\texttt{any} \leq \texttt{any}}\ (\text{ST\_ANY\_REFL})$$

que permite preservar las propiedades de orden. Esto se hace de la misma forma que en la Sección 4 de Siek and Taha [2007][10], justificado con la idea de que el tipo dinámico any debe ser neutral respecto al subtipado[11].

En el artículo antes referido se propone una nueva relación, acuñada *subtipado consistente*, con el fin de integrar de una manera correcta ambas relaciones (consistencia y subtipado) y así reemplazar al subtipado en la extensión de su sistema de tipos.

Formalmente, el subtipado consistente $\lesssim$ se define como la clausura transitiva de la unión de las relaciones $\leq$ y $\sim$ (habitualmente escrito como $(\leq \cup \sim)^*$). Aquí se presenta explícitamente por un conjunto de reglas, que incluyen todos los casos de las reglas para el subtipado (cambiando $\leq$ por $\lesssim$) además de los tres casos base de la relación de consistencia:

$$\frac{}{\texttt{any} \lesssim \texttt{any}}\ (\text{SC\_ANY\_REFL}) \qquad \frac{}{\texttt{any} \lesssim \tau}\ (\text{SC\_ANY\_L}) \qquad \frac{}{\tau \lesssim \texttt{any}}\ (\text{SC\_ANY\_R})$$

La definición completa de las reglas se encuentra en el Anexo C.2.2. Algunos ejemplos a continuación:

$$\texttt{\{any,integer\}} \lesssim \texttt{\{atom,number\}},\ \texttt{\%\{1 => integer,2 => any\}} \lesssim \texttt{\%\{1 => number\}}$$

$$\texttt{\{number,any\}} \not\lesssim \texttt{\{integer,any\}},\ \texttt{[any]} \not\lesssim \texttt{[]},\ \texttt{\%\{\}} \not\lesssim \texttt{\%\{1 => any\}}$$

Es importante mencionar que el subtipado consistente no es un preorden porque no es una relación transitiva. Por ejemplo:

$$\texttt{\{atom,atom\}} \lesssim \texttt{\{any,any\}} \lesssim \texttt{\{integer,number\}}$$

mientras que

$$\texttt{\{atom,atom\}} \not\lesssim \texttt{\{integer,number\}}$$

3.2.2.4. Relación de subtipado precisión

Es la relación análoga al subtipado consistente pero con respecto a la precisión ($\lll$) en vez de la consistencia, en cuanto se define formalmente como $\prec^{-}= (\leq \cup \lll)^*$.

La definición inductiva usando un conjunto de reglas dirigidas por sintaxis es algo más complicado que en el caso anterior y se define por mutua recursión con las de la relación de *subtipado materialización*[12] $\prec^{+}= (\leq \cup \ggg)^*$ en el Anexo C.4.

Será de interés tener en consideración el siguiente resultado -no probado aún- para una discusión posterior:

Las relaciones de subtipado precisión y subtipado materialización son ordenes en $T/\leq$: La relación de $\prec^{-}$ es reflexiva y transitiva. Además, a menos de la equivalencia inducida por la clausura estructural de la relación de permutación entre mapas, $\prec^{-}$ es antisimétrica. Lo mismo para $\prec^{+}$.

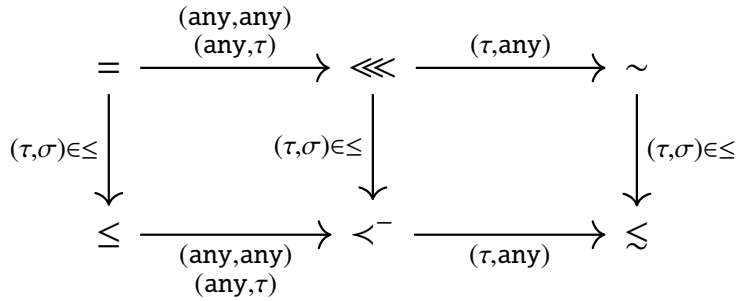
[10] En ese trabajo, se refiere a la relación de consistencia y la de subtipado como "ortogonales". Si se piensa al subtipado en T como un conjunto ordenado, entonces la extensión "ortogonal" hacia $\tilde{T}$ será la pequeña posible dentro de los conjuntos ordenados; ésta relación es exáctamente la que se consigue añadiendo la regla ST_ANY_REFL.

[11] En contraste con trabajos anteriores que trataban al tipo any como un tipo máximo del retículo, llevando a inconsistencias en el sistema de subtipado

[12] Se toma el término materialización de Castagna et al. [2019]; la definición equivale a $\lll^{op}$ ($\ggg$)

3.2.2.5. Resumen de las relaciones presentadas

Para fines didácticos, se presenta a continuación un diagrama que unifica todas las relaciones que han sido introducidas en esta sección.



La igualdad de tipos (=) representa la relación reflexiva que relaciona a cada $\tau \in \tilde{T}$ consigo mismo y nada más. Cada otro nodo del diagrama representa una de las relaciones introducidas en ésta sección y la etiqueta sobre cada flecha indica los nuevos casos que se incluyen en el nodo de la cola para formar el nodo de la cabeza. En lenguaje matemático, se puede decir que $\mathbf{head}(arr) = (\mathbf{tail}(arr) \cup \mathbf{label}(arr))^*$ para cada flecha arr del diagrama.

También es válido un diagrama similar reemplazando $\prec^-$ por $\prec^+$ al intercambiar el rol de (any, τ) con (τ, any).

3.2.3. Generalización del supremo e ínfimo

A pesar de que el subtipado consistente constituye un candidato fructífero para reemplazar al subtipado en un contexto gradual, no es una relación de orden, por lo que no sugiere una manera razonable de extender al supremo en el contexto gradual.

3.2.3.1. Semántica AGT

En el artículo Garcia et al. [2016] titulado "Abstracting Gradual Semantics", se propone una semántica novel para los tipos graduales -aludida como AGT, de aquí en adelante- basada en una interpretación hacia conjuntos de tipos estáticos. Explícitamente, se define la función de concretización $\gamma : \tilde{T} \to \mathcal{P}(T)$ que asigna, a cada $\tau \in \tilde{T}$, el conjunto de "concretizaciónes" $\{\sigma \in T : \sigma \lll \tau\}$. Esta definición se utiliza como punto de partida en el diseño de un mecanismo que permite extender predicados $P(T, T)$ y funciones $F(T, T) = T$ sistemáticamente hacia predicados $\tilde{P}(\tilde{T}, \tilde{T})$ y funciones $\tilde{F}(\tilde{T}, \tilde{T}) = \tilde{T}$ entre tipos graduales.

La metodología de extensión de AGT prueba su utilidad al lograr, por ejemplo, recuperar el subtipado consistente $\lesssim$ como la extensión gradual $\tilde{\lesssim}$ del subtipado estático (ver **Proposition 13** de ese artículo). Más aún, demuestra que la extensión del supremo $\tilde{\vee}$ es un reemplazo exitoso de $\vee$ a efectos de la construcción de un sistema de tipos algorítmico basado en subtipado estructural[13].

[13] Sienta la base para la definición de un lenguaje intrínsecamente gradual donde las propiedades formales fundamentales de los sistemas graduales se prueban en términos semánticos

3.2.3.2. Algoritmo del supremo gradual a partir de AGT

La definición explícita del supremo/ínfimo gradual basada en AGT adaptada para este trabajo se encuentra en el Recuadro 3.3 e incluye todos los casos de la versión estática, además de agregar nuevos casos para cuando alguno de los operandos resulta ser `any`.

El algoritmo se basa sin supuestos extra en la extensión AGT de $\vee$, que identifica $\tau \tilde{\vee} \sigma$ como el tipo $\mu \in \tilde{T}$ que supone la mejor aproximación posible vía γ[14] del conjunto $\{\tau_0 \vee \sigma_0 : \tau_0 \in \gamma(\tau), \sigma_0 \in \gamma(\sigma)\}$ formado por todos los posibles supremos de las concretizaciones de τ y σ. A continuación se detalla la justificación para algunas instancias particulares:

- `integer` $\vee$ `any` = `any`: Los σ para los cuales `integer` $\vee \sigma$ está definido son `integer`, `float` y `number`, de los cuales puede resultar `integer` $\vee \sigma$ = `integer` (si σ = `integer`) o `integer` $\vee \sigma$ = `number` (si $\sigma \in \{$`float`, `number`$\}$). La mejor aproximación del conjunto de supremos posibles {`integer`, `number`} vía γ es `any`, ya que γ(`any`) = $T \supseteq \{$`integer`, `number`$\}$ y en éste caso cualquier $\mu \in \tilde{\tau}$ distinto de `any` cumple {`integer`, `number`} $\not\subseteq \gamma(\mu)$
- `number` $\vee$ `any` = `number`: Los σ para los cuales `number` $\vee \sigma$ está definido son `integer`, `float` y `number`, de los cuales puede resultar únicamente `number` $\vee \sigma$ = `number` (pues `number` es maximal). El conjunto de supremos posibles es {`number`}, luego está aproximado vía γ (de hecho es igual) por `number` (ya que γ(`number`) = {`number`} por ser estático.
- `[`τ`]` $\tilde{\vee}$ `any` = `[`$\tau \tilde{\vee}$ `any]`: Los σ para los cuales `[`τ`]` $\tilde{\vee} \sigma$ está definido son `[ ]` y los tipos de la forma `[`σ_0`]` con $\tau \tilde{\vee} \sigma_0$ definido. En el primer caso `[`τ`]` $\tilde{\vee} \sigma$ = `[`τ`]` y en el segundo `[`τ`]` $\tilde{\vee} \sigma$ = `[`$\tau \tilde{\vee} \sigma_0$`]` (que ya incluye al primero con $\sigma_0 = \tau$). Luego la mejor aproximación del conjunto de supremos será `[`μ`]` donde μ es la mejor aproximación vía γ del conjunto $\{\tau_0 \vee \sigma_0 : \tau_0 \in \gamma(\tau), \sigma_0 \in T\} = \{\tau_0 \vee \sigma_0 : \tau_0 \in \gamma(\tau), \sigma_0 \in \gamma($`any`$)\}$, es decir, $\mu = \tau \vee$ `any`, de lo cual se concluye lo que se buscaba.

El algoritmo GSup depende de dos predicados Maximal y Minimal, que chequean cuando un tipo en B es maximal o minimal.

- Maximal devuelve **true** con `number` y `atom` y **false** en el resto
- Minimal devuelve **true** con `integer`, `float` y cada átomo a y **false** en el resto

3.2.3.3. Supremo arbitrario

Para pasar del supremo binario $\tau \tilde{\vee} \sigma$ al supremo arbitrario $\widetilde{\bigvee}_{i=1}^{n} \tau_i$ se podría imitar ingenuamente el mismo procedimiento que se utilizó para definir al supremo original, i.e., reducirlo a la iteración del caso binario. Lamentablemente, esto no funciona correctamente, como se puede observar en el siguiente ejemplo:

$$\texttt{\{integer,any\}} \tilde{\vee} (\texttt{\{float,integer\}} \tilde{\vee} \texttt{\{any,float\}}) = \texttt{\{any,number\}}$$

$$(\texttt{\{integer,any\}} \tilde{\vee} \texttt{\{float,integer\}}) \tilde{\vee} \texttt{\{any,float\}} = \texttt{\{number,any\}}$$

mientras que con la metodología de AGT (generalizado para un predicado de aridad 3) se infiere:

$$\widetilde{\bigvee} \{\texttt{\{integer,any\}}, \texttt{\{float,integer\}}, \texttt{\{any,float\}}\} = \texttt{\{number,number\}}$$

[14] Esto se hace encontrando un adjunto inferior para γ en el retículo $(\mathcal{P}(T), \subseteq)$. Se pueden ver los detalles en el artículo.

No parece inalcanzable intentar generalizar el algoritmo GSup para lograr la forma determinada por AGT, sin embargo este objetivo ha quedado fuera del trabajo. Por lo pronto, esto impacta en la extensión gradual de T_CASE: mientras que la regla será formulada en términos de $\widetilde{\bigvee}_{i=1}^{n}\tau_i$, en la implementación actual se suplanta por $(...(\tau_1 \tilde{\vee} \tau_2)\tilde{\vee}...)\tilde{\vee}\tau_n)$.

procedure GSup(τ, σ, b)
  switch (τ, σ, b) **do**
    case (τ, any, **true**) **when** $\tau \in B$ **and** Maximal(τ)
      return τ
    case (τ, any, **true**) **when** $\tau \in B$ **and not** Maximal(τ)
      return any
    case (τ, any, **false**) **when** $\tau \in B$ **and** Minimal(τ)
      return τ
    case (τ, any, **false**) **when** $\tau \in B$ **and not** Minimal(τ)
      return any
    case ([ ], any, **true**)
      return any
    case ([ ], any, **false**)
      return [ ]
    case ([τ], any, **true**)
      return [GSup(any, τ)]
    case ([τ], any, **false**)
      return [any]
    case ({τ_1 , ... , τ_n}, any, b)
      return {GSup(τ_1, any, b) , ... , GSup(τ_n, any, b)}
    case (%{$\overline{(k_i \Rightarrow \tau_i)_{i\in I}}$}, any, b)
      return %{$\overline{(k_i \Rightarrow \text{GSup(any}, \tau_i, b))_{i\in I}}$}
    case ($(\tau_1, .., \tau_n) \to \tau_0$, any, b)
      $nb \leftarrow$ **not** b
      return (GSup(τ_1, any, nb), .., GSup(τ_n, any, nb)) $\to$ GSup(τ_0, any$_0$, b)
    case (any, τ, b)
      return GSup(τ, any, b)
    $\cdots$ // el resto de casos es igual al de Sup
end procedure

Recuadro 3.3: Algoritmo de supremo e ínfimo gradual

3.2.4. Extensión del sistema de tipos

Considerando las relaciones definidas entre tipos graduales, los cambios requeridos para gradualizar el sistema de tipos son menores. A continuación se discuten los cambios en detalle; el sistema gradual se puede ver completo en los Anexos C.6 y C.7.

3.2.4.1. Cambios en el chequeo de un programa

Una de las motivaciones para la elección de un sistema gradual frente a uno estático es la flexibilidad que se brinda al usuario del lenguaje para calibrar la cantidad de información de tipos que desea incluir en sus programas. En este trabajo, el punto anterior se ve reflejado en que:

1. Se admite al tipo dinámico any dentro de la sintaxis de los tipos de las especificaciones
2. Se admiten programas parcialmente especificados

En cuanto al punto 1., las ocurrencias de any en las especificaciones no suponen ningún cambio al nivel del chequeo de las declaraciones, a excepción de cambiar el requerimiento $\sigma \leq \tau_0$ en T_DEF por $\sigma \lesssim \tau_0$:

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \emptyset;\emptyset;\tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset;\Gamma_{n-1};\tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta;\Gamma_n \vdash^{t} e : \sigma \Rightarrow \Gamma' \quad \sigma \lesssim \tau_0\end{array}}{\Delta \vdash^{t} \texttt{def}\ \ f\_name(p_1, \ldots, p_n)\ \ \texttt{do}\ \ e\ \ \texttt{end}}\ \text{(T\_DEF)}$$

Para hacer frente al punto 2., el sistema incluye la nueva regla T_DEF_ANY que simula la especificación dinámica (any, .., any) → any de la aridad que corresponda. Esta es una diferencia importante con el sistema de Cassola et al. [2022], donde la regla para este caso consistía en omitir cualquier tipo de chequeo sobre la función en cuestión.

$$\frac{\begin{array}{c}(f\_name, n) \notin \mathrm{dom}(\Delta) \\ \emptyset;\emptyset;\texttt{any} \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset;\Gamma_{n-1};\texttt{any} \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta;\Gamma_n \vdash^{t} e : \sigma \Rightarrow \Gamma'\end{array}}{\Delta \vdash^{t} \texttt{def}\ \ f\_name(p_1, \ldots, p_n)\ \ \texttt{do}\ \ e\ \ \texttt{end}}\ \text{(T\_DEF\_ANY)}$$

Notar que el chequeo $\sigma \lesssim \tau_0$ en T_DEF correspondería en T_DEF_ANY a $\sigma \lesssim \texttt{any}$, pero gracias a SC_ANY_R, esta condición siempre se cumple, por lo que resulta innecesario incluirla como hipótesis.
Lejos de ser trivial, la regla T_DEF_ANY permite evitar errores de chequeo dentro del cuerpo de e, aunque no exista ninguna restricción de tipo ni para los parámetros ni para el retorno. Por ejemplo, el módulo

```
defprogram Program do
    @spec sum(number, number) :: number
    def sum(x, y) do
        x + y
    end

    def main() do
        sum(1, :true)
    end
end
```

se rechaza porque el cuerpo de `main` no tipa correctamente.

El cambio respecto a Cassola et al. [2022] tiene la gran desventaja de que no admite la compatibilidad con código legado, ya que cualquier expresión debe ser chequeable respecto al sistema actual. Sin embargo, el cambio es necesario para garantizar que las funciones anotadas recibirán recibirán argumentos correctos para sus parámetros. Por ejemplo, el programa

```
defprogram Program do
    @spec sum(number, number) :: number
    def sum(x, y) do
        x + y
    end

    def untyped(x) do
        x
    end

    def main() do
        sum(1, untyped(:true))
    end
end
```

se chequea correctamente pero -según se verá en el capítulo siguiente- el chequeo se utiliza para añadir un cast hacia `number` en el segundo parámetro de la llamada de `sum/2` para evitar su invocación con tipos incorrectos.

3.2.5. Cambios en el chequeo-refinamiento de patrones

Un cambio fundamental en las reglas de chequeo de expresiones consiste en el reemplazo de todas las ocurrencias del subtipado ($\leq$), supremo ($\vee$) e ínfimo ($\wedge$) por sus versiones graduales ($\lesssim, \tilde{\vee}, \tilde{\wedge}$). Esto permite, por ejemplo, validar la secuencia:

```
1    x # tipa como any
2    x = 1
```

donde la derivación que corresponde al juicio de colección de variables para la segunda línea es

$$\dfrac{\dfrac{type(1) = \texttt{integer} \lesssim \texttt{any}}{[x \mapsto \texttt{any}]; \emptyset; \texttt{any} \vdash^{pc} \texttt{x} \Rightarrow [x \mapsto \texttt{integer}]} \qquad [x \mapsto \texttt{any}]; [x \mapsto \texttt{integer}]; \texttt{any} \vdash^{ptv} \texttt{x} : \texttt{integer}}{[x \mapsto \texttt{any}]; \texttt{any} \vdash^{pt} \texttt{x} : \texttt{integer} \Rightarrow [x \mapsto \texttt{integer}]}$$

Otros ejemplos de nuevos juicios validados son:

$$\emptyset; \texttt{\{integer,any\}} \vdash^{pt} \texttt{\{x,x\}} : \texttt{\{integer,integer\}} \Rightarrow [x \mapsto \texttt{integer}]$$
$$\emptyset; \texttt{\{any,float\}} \vdash^{pt} \texttt{\{x,x\}} : \texttt{\{float,float\}} \Rightarrow [x \mapsto \texttt{float}]$$
$$\emptyset; \texttt{\{number,any\}} \vdash^{pt} \texttt{\{x,x\}} : \texttt{\{any,any\}} \Rightarrow [x \mapsto \texttt{any}]$$

En los dos primeros juicios el resultado es satisfactorio, ya que se propaga el tipo de la primera componente a la segunda para incrementar la información estática del tipo inferido.

Sin embargo, el tercer juicio "olvida" el hecho de que la primera coordenada del tipo candidato es `number`, con lo que el tipo otorgado en el juicio (`{any,any}`) es más impreciso que el original (`{number,any}`). Esto se debe a que `number` $\tilde{\wedge}$ `any` = `any`; y en general, al hecho que el supremo consistente no proviene de la precisión (y por lo tanto, $\mu = \tau \wedge \sigma$ no implica $\mu \lll \tau, \sigma$).

El segundo cambio consiste en el añadido de dos nuevas reglas TPC_GROUND y TPV_GROUND para los casos en que el tipo candidato es `any`:

$$\frac{\mathit{ground}_P(p) = \tau \qquad \Sigma; \Gamma; \tau \vdash^{pc} p \Rightarrow \Gamma'}{\Sigma; \Gamma; \texttt{any} \vdash^{pc} p \Rightarrow \Gamma'} \text{ (TPC\_GROUND)}$$

$$\frac{\mathit{ground}_P(p) = \tau \qquad \Sigma; \Gamma; \tau \vdash^{ptv} p : \sigma}{\Sigma; \Gamma; \texttt{any} \vdash^{ptv} p : \sigma} \text{ (TPV\_GROUND)}$$

donde la definición de $\mathit{ground}_P \in P \rightharpoonup G$ corresponde al procedimiento GroundP(p):

procedure GroundP(p)
  switch p **do**
    case l : **return** Type(l)
    case `[ ]` : **return** `[ ]`
    case `[`p`]` : **return** `[any]`
    case `{`p_1`,...,`p_n`}` : **return** `{any,...,any}`
    case `%{`k_1 `=>` p_1`,...,`k_n `=>` p_n`}` : `%{`k_1 `=>` `any,...,`k_n `=>` `any}`
    default : **fail**
end procedure

Las dos nuevas reglas son necesarias para poder establecer la validez de ciertos casos como

$$\emptyset; \texttt{any} \vdash^{pt} \texttt{\{x,y\}} : \texttt{\{any,any\}} \Rightarrow [x \mapsto \texttt{any}, y \mapsto \texttt{any}]$$
$$\emptyset; \texttt{\{any,:a\}} \vdash^{pt} \texttt{\{\{x,y\},x\}} : \texttt{\{\{:a,any\},:a\}} \Rightarrow [x \mapsto \texttt{:a}]$$

en los que el sistema de Cassola et al. [2022] recurría indirectamente a la regla T_DOWN. En ese sentido, la justificación es similar al cambio en la regla TPC_MAP a partir de la eliminación de la regla T_SUB.

Por ejemplo, la derivación para el juicio

$$\emptyset; \texttt{\{any,:a\}} \vdash^{pt} \texttt{\{\{x,y\},x\}} : \texttt{\{\{:a,any\},:a\}} \Rightarrow [x \mapsto \texttt{:a}]$$

se compone de las dos derivaciones

$$\dfrac{\dfrac{\dfrac{\vdots}{\emptyset; \emptyset; \texttt{\{any,any\}} \vdash^{pc} \texttt{\{x,y\}} \Rightarrow [x \mapsto \texttt{any}, y \mapsto \texttt{any}]}}{\emptyset; \emptyset; \texttt{any} \vdash^{pc} \texttt{\{x,y\}} \Rightarrow [x \mapsto \texttt{any}, y \mapsto \texttt{any}]} \qquad \dfrac{\texttt{any}\tilde{\wedge}\texttt{:a} = \texttt{:a}}{\emptyset; [x \mapsto \texttt{any}, y \mapsto \texttt{any}]; \texttt{any} \vdash^{pc} \texttt{x} \Rightarrow [x \mapsto \texttt{:a}, y \mapsto \texttt{any}]}}{\emptyset; \emptyset; \texttt{\{any,:a\}} \vdash^{pc} \texttt{\{\{x,y\},x\}} \Rightarrow [x \mapsto \texttt{:a}, y \mapsto \texttt{any}]}$$

$$\dfrac{\dfrac{\vdots}{\emptyset; [x \mapsto \texttt{:a}, y \mapsto \texttt{any}]; \{\texttt{any},\texttt{any}\} \vdash^{ptv} \{\texttt{x},\texttt{y}\} : \{\texttt{:a},\texttt{any}\}}}{\emptyset; [x \mapsto \texttt{:a}, y \mapsto \texttt{any}]; \texttt{any} \vdash^{ptv} \{\texttt{x},\texttt{y}\} : \{\texttt{:a},\texttt{any}\}} \qquad \dfrac{\texttt{any} \tilde{\wedge} \texttt{:a} = \texttt{:a}}{\emptyset; [x \mapsto \texttt{:a}, y \mapsto \texttt{any}]; \texttt{any} \vdash^{pc} \texttt{x} \Rightarrow \texttt{:a}}}{\emptyset; [x \mapsto \texttt{:a}, y \mapsto \texttt{any}]; \{\texttt{any},\texttt{:a}\} \vdash^{ptv} \{\{\texttt{x},\texttt{y}\},\texttt{x}\} : \{\{\texttt{:a},\texttt{any}\},\texttt{:a}\}}$$

Finalmente, se espera en un trabajo futuro poder generalizar la propiedad de refinamiento de tipo al contexto gradual:

Refinamiento consistente de tipo en el chequeo de patrones: Cada vez que el juicio $\Sigma; \Gamma \vdash^{pt} \tau : p \Rightarrow \sigma$ es correcto, vale que $\sigma \lesssim \tau$. Más aún, si además p no tiene variables repetidas, $\sigma \sim \tau$.

3.2.6. Cambios en el chequeo de una expresión

Al igual que en el chequeo de patrones, en el sistema de chequeo de expresiones se reemplazan todas las ocurrencias del subtipado ($\leq$) y el supremo ($\vee$) por sus versiones graduales ($\lesssim$, $\tilde{\vee}$).

Así, por ejemplo, la regla T_IFE luce idéntica a menos de esos reemplazos:

$$\dfrac{\begin{array}{c}\Delta; \Gamma \vdash^{t} e : \tau \Rightarrow \Gamma' \quad \tau \lesssim \texttt{boolean} \\ \Delta; \Gamma \dagger \Gamma' \vdash^{t} e_1 : \tau_1 \Rightarrow \Gamma_1 \quad \Delta; \Gamma \dagger \Gamma' \vdash^{t} e_2 : \tau_2 \Rightarrow \Gamma_2 \quad \tau_1 \tilde{\vee} \tau_2 = \sigma\end{array}}{\Delta; \Gamma \vdash^{t} \texttt{if}\ e\ \texttt{do}\ e_1\ \texttt{else}\ e_2\ \texttt{end} : \sigma \Rightarrow \Gamma'}\ \text{(T\_IFE)}$$

La siguiente instancia de T_IFE ilustra el impacto práctico de los cambios en dicha regla, tomando $\Gamma = [b \mapsto \texttt{any}, i \mapsto \texttt{integer}, n \mapsto \texttt{number}, x \mapsto \texttt{any}]$:

$$\dfrac{\begin{array}{c}\emptyset; \Gamma \vdash^{t} \texttt{b} : \texttt{any} \Rightarrow \emptyset \quad \texttt{any} \lesssim \texttt{boolean} \\ \emptyset; \Gamma \vdash^{t} \texttt{\{i,n\}} : \texttt{\{integer,number\}} \Rightarrow \emptyset \quad \Delta; \Gamma \vdash^{t} \texttt{x} : \texttt{any} \Rightarrow \emptyset \\ \texttt{\{integer,number\}} \tilde{\vee} \texttt{any} = \texttt{\{any,number\}}\end{array}}{\emptyset; \Gamma \vdash^{t} \texttt{if b do \{i,n\} else x end} : \texttt{\{any,number\}} \Rightarrow \emptyset}$$

Se observa:

1. La variable `b` es de tipo `any`, y es válida como condición porque `any` $\sim$ `boolean`.
2. La primera coordenada del tipo otorgado a la expresión es `any` ya que podría variar entre `integer` y `number` en caso de estar definido.
3. La segunda coordenada del tipo otorgado a la expresión es `number` ya que no se admite otra opción en caso de estar definido.

Otro cambio en las reglas de chequeo consiste en gradualizar las funciones de tipos $uniOpType(\ominus, \cdot, \cdot) : T \rightharpoonup T$ y $binOpType(\oplus, \cdot, \cdot) : T \times T \rightharpoonup T$ hacia $\widetilde{uniOpType}(\ominus, \cdot, \cdot) : \tilde{T} \rightharpoonup \tilde{T}$ y $\widetilde{binOpType}(\oplus, \cdot, \cdot) : \tilde{T} \times \tilde{T} \rightharpoonup \tilde{T}$ siguiendo la metodología AGT.

Adaptada a al contexto de este trabajo, la gradualización AGT de una función a nivel de tipos se materializa de la siguiente forma para cualquiera de los procedimientos presentados en el Anexo B.4:

- Si $\tau, \sigma \in B$ utilizar el procedimiento original.
- Si $\tau = \texttt{any}, \sigma \in B$, el resultado es `any` si el conjunto de resultado posibles $\{\mu_1, ..., \mu_n\}$ variando $\tau \in B$ tiene más de un elemento, y μ_1 si es un solo elemento.
- Si $\tau \in B, \sigma = \texttt{any}$, el resultado es `any` si el conjunto de resultado posibles $\{\mu_1, , ..., \mu_n\}$ variando $\sigma \in B$ tiene más de un elemento, y μ_1 si es un solo elemento.

- Si $\tau = \sigma = \texttt{any}, \sigma \in B$, el resultado es any si el conjunto de resultado posibles $\{\mu_1, , ..., \mu_n\}$ variando $\tau, \sigma \in B$ tiene más de un elemento, y μ_1 si es un solo elemento.

Por ejemplo, la gradualización de *sumType* se describe al modificar el procedimiento SumType según la descripción anterior:

procedure GSumType(τ, σ)
  switch (τ, σ) **do**
    case (float, any)
      return float
    case (any, float)
      return float
    case (τ, any) **when** $\tau \leq$ number
      return any
    case (any, σ) **when** $\sigma \leq$ number
      return any
    default
      SumType(τ, σ)
end procedure

La gradualización del resto de funciones de tipos para los operadores del fragmento se detalla en el Anexo C.5.

Usando la definición anterior para $\widetilde{sumType}$, se logra validar:

$$\frac{\emptyset; [i \mapsto \texttt{integer}, x \mapsto \texttt{any}] \vdash^t \texttt{i} : \texttt{integer} \Rightarrow \emptyset \quad \emptyset; [i \mapsto \texttt{integer}, x \mapsto \texttt{any}] \vdash^t \texttt{x} : \texttt{any} \Rightarrow \emptyset \quad \widetilde{sumType}(\texttt{integer}, \texttt{any}) = \texttt{any}}{\emptyset; [i \mapsto \texttt{integer}, x \mapsto \texttt{any}] \vdash^t \texttt{i+x} : \texttt{any} \Rightarrow \emptyset}$$

$$\frac{\emptyset; [f \mapsto \texttt{float}, x \mapsto \texttt{any}] \vdash^t \texttt{f} : \texttt{float} \Rightarrow \emptyset \quad \emptyset; [f \mapsto \texttt{float}, x \mapsto \texttt{any}] \vdash^t \texttt{x} : \texttt{any} \Rightarrow \emptyset \quad \widetilde{sumType}(\texttt{float}, \texttt{any}) = \texttt{float}}{\emptyset; [f \mapsto \texttt{float}, x \mapsto \texttt{any}] \vdash^t \texttt{f+x} : \texttt{float} \Rightarrow \emptyset}$$

La asimetría en ambos casos se debe a que las dos "concretizaciones" de la derivación de la derecha resultan en el tipo float, mientras que el de la izquierda varía en {integer, float}.

Para el caso de las llamadas anónimas, se agrega una reglas que considera el caso en que la expresión llamada es de tipo any:

$$\frac{\Delta; \Gamma \vdash^t e : \texttt{any} \Rightarrow \Gamma_0 \quad \Delta; \Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta; \Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n}{\Delta; \Gamma \vdash^t e\texttt{.(}e_1\texttt{,...,}e_n\texttt{)} : \texttt{any} \Rightarrow \Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (T\_ACALL\_ANY)}$$

Esta regla permite utilizar el tipo dinámico any como el una función de aridad n, y está validada implícitamente por el hecho que $\texttt{any} \sim (\texttt{any}, .., \texttt{any}_n) \to \texttt{any}$.

Para finalizar con la gradualización de las reglas de chequeo de expresiones, se incluyen dos reglas adicionales que tratan los casos que involucran declaraciones de función no especificadas. Nuevamente, éstas simulan la especificación dinámica de la aridad correspondiente:

$$\frac{(f\_name, n) \notin \text{dom}(\Delta)}{\Delta; \Gamma \vdash^t \texttt{\&}f\_name\texttt{/}n : (\texttt{any}_1, .., \texttt{any}_n) \to \texttt{any} \Rightarrow \emptyset} \text{ (T\_ANON\_ANY)}$$

$$\frac{(f\_name, n) \notin \mathrm{dom}(\Delta) \qquad \Delta;\Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^t e_n : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^t f\_name(e_1, ..., e_n) : \mathsf{any} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{(T\_DCALL\_ANY)}$$

Observar que con estas reglas no se corre riesgo de validar expresiones que contengan funciones no declaradas, puesto que esto ya lo resuelve el compilador.

3.2.7. Conservatividad respecto al sistema estático

La versión gradual del sistema constituye una extensión -en el sentido formal de la palabra- del sistema estático de la sección anterior, ya que conserva la validez de todos sus juicios, y en particular, el de chequeo de programas. Este último hecho se puede expresar formalmente por:

Conservatividad del juicio estático de chequeo de programas respecto al sistema gradual: Si $\vdash^t m$ en el sistema estático, entonces $\vdash^t m$ en el sistema gradual.

Si bien la prueba exhaustiva de este enunciado ha quedado fuera del alcance del proyecto, ésta debería seguirse sin dificultades de las siguientes observaciones:

1. Los predicados y funciones a nivel de tipos que ocurren en el sistema ($\lesssim$, $\tilde{\vee}$, $\tilde{\wedge}$, $\widetilde{type}$, $\widetilde{uniOpType}$, $\widetilde{binOpType}$) se gradualizaron de acuerdo a la metodología AGT, y ésta asegura que conservan el comportamiento original si se restringen a tipos estáticos.
2. Todas las reglas del sistema gradual que provienen del sistema estático coinciden con la versión original cuando se restringen a tipos estáticos, ya que sus cambios involucran únicamente la gradualización de predicados y funciones según AGT y siendo que estas son conservativas respecto al sistema estático como se establece en el punto anterior.

3.2.8. Validación de la Static Gradual Guarantee

En Siek et al. [2015] se introduce la *Gradual Guarantee*, que comprende un listado de criterios (o "garantías") que -según se argumenta- deberían acatar los diseñadores de sistemas graduales, para permitir una transición "predecible" de un código no tipado a uno tipado. Si bien tienen su formulación matemática concreta, a grandes rasgos dichos criterios establecen que todos los cambios en un programa que resulten *únicamente* de remover precisión en las anotaciones de tipos (esto es, volverlo "más dinámico") no deberían modificar su semántica, a excepción de posibilitar nuevos errores debido a inconsistencias en runtime.

En particular, se distingue la *Static Gradual Guarantee* -de aquí en más se abreviará como SGG- que adaptado al contexto actual, establece que si cierto programa m válido respecto al sistema de tipos pierde precisión -transformándose en m'-, entonces m' también será válido. Formalmente:

Static Gradual Guarantee: Si $\vdash^t m$ y $m \lll m'$, entonces $\vdash^t m'$.

Para el sistema presentado en este capítulo, el criterio se satisface, y su prueba depende de los siguientes lemas auxiliares:

Static Gradual Guarantee (PROG): Si $\vdash^c m \Rightarrow \Delta$, $\vdash^c m' \Rightarrow \Delta'$, $\Delta \vdash^{ts} m$ y $m \lll m'$, entonces $\Delta \lll \Delta'$ y $\Delta' \vdash^{ts} m'$.
Static Gradual Guarantee (DEF): Si $\Delta \vdash^t d$, $\Delta \lll \Delta'$ y $d \lll d'$, entonces $\Delta' \vdash^t d'$.

Static Gradual Guarantee (EXPR): Si $\Delta;\Gamma_1 \vdash^t e : \tau \Rightarrow \Gamma_2$, $\Delta \lll \Delta'$ y $\Gamma_1 \lll \Gamma_1'$, entonces $\Delta';\Gamma_1' \vdash^t e : \tau' \Rightarrow \Gamma_2', \tau \lll \tau'$ y $\Gamma_2 \lll \Gamma_2'$.
Static Gradual Guarantee (PAT1): Si $\Sigma;\Gamma_1;\tau \vdash^{pc} p \Rightarrow \Gamma_2$, $\Sigma \lll \Sigma'$, $\Gamma_1 \lll \Gamma_1'$ y $\tau \lll \tau'$ entonces $\Sigma';\Gamma_1';\tau' \vdash^{pc} p \Rightarrow \Gamma_2'$ y $\Gamma_2 \lll \Gamma_2'$.
Static Gradual Guarantee (PAT2): Si $\Sigma;\Gamma;\tau \vdash^{ptv} p : \sigma, \Sigma \lll \Sigma', \Gamma \lll \Gamma'$ y $\tau \lll \tau'$ entonces $\Sigma';\Gamma';\tau' \vdash^{ptv} p : \sigma'$ y $\sigma \lll \sigma'$.

Los lemas se prueban en orden opuesto a como se listaron. A continuación se bosqueja la demostración; el argumento se basa en Garcia et al. [2016] (**Proposition 11**):

1. Todos los lemas manifiestan la monotonía de los distintos juicios respecto a $\lll$, partiendo de las posiciones *input* hacia las *output*.
2. La gradualización de los predicados y funciones de tipos auxiliares en las reglas que validan los juicios en el sistema gradual son exactamente las gradualizaciones según AGT de las versiones originales.
3. Todos los predicados y funciones gradualizados según AGT respetan la monotonía respecto a $\lll$.
4. Los casos $\mathsf{any} \lll \tau$ en PAT1 Y PAT2 donde se discute según la forma de τ (TPC_CONS, TPC_TUPLE, TPC_MAP, TPC_LIST y análogos de refinamiento) se prueban gracias al uso de TPC_GROUND y TPV_GROUND respectivamente.
5. Los casos de reglas nuevas (T_DEF, TPC_GROUND, TPV_GROUND, T_ACALL_ANY, T_ANON_ANY, T_DCALL_ANY) se verifican inductivamente.
6. Para cada caso de DEF, PROG y EXPR que involucra el contexto de declaraciones Δ -T_DEF, T_ANON, T_DCALL-, la superprecisión $\bar{\lll}$ se justifica con las reglas que simulan la especificación dinámica (T_DEF_ANY, T_ANON_ANY, T_DCALL_ANY)
7. El resto de los casos se sigue por inducción y usando los lemas previamente probados.

El argumento detallado de los puntos 4., 5. y 6. se encuentra en el Anexo C.9.

3.2.9. Comparación con sistemas basados en subsumption

Mientras que la extensión gradual de esta sección se basa en la metodología AGT, el sistema de Cassola et al. [2022] se basa en Castagna et al. [2019], al introducir la regla T_DOWN (allí Materialize) que cumple el rol análogo a la regla de subsumption T_SUB para la relación opuesta a la precisión ($\ggg$). Explícitamente, las reglas consideradas son:

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \tau \leq \sigma}{\Delta;\Gamma \vdash^t e : \sigma \Rightarrow \Gamma'}\ \text{(T\_SUB)} \qquad \frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \tau \ggg \sigma}{\Delta;\Gamma \vdash^t e : \sigma \Rightarrow \Gamma'}\ \text{(T\_DOWN)}$$

Notar que ambas reglas pueden condensarse en una sola que considera la relación de subtipado-materialización $\prec^+ = (\leq \cup \ggg)^*$:

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \tau \prec^+ \sigma}{\Delta;\Gamma \vdash^t e : \sigma \Rightarrow \Gamma'}\ \text{(T\_SUBDOWN)}$$

Dado que el sistema con T_SUBDOWN es declarativo, una vía posible para analizar las diferencias con el sistema actual consiste en aplicar la metodología de Pierce para definir un sistema algorítmico equivalente a ese sistema. Para el sistema resultante, la diferencia clave respecto al de este trabajo es que el supremo involucrado es el que proviene del subtipado-materialización $\bigvee^+$ y no el de AGT $\widetilde{\bigvee}$. A partir de este cambio se consigue, por ejemplo:

$$\frac{\emptyset;\Gamma \vdash^t \texttt{b} : \texttt{any} \Rightarrow \emptyset \quad \texttt{any} \prec^+ \texttt{boolean} \quad \emptyset;\Gamma \vdash^t \texttt{\{i,n\}} : \texttt{\{integer,number\}} \Rightarrow \emptyset \quad \Delta;\Gamma \vdash^t \texttt{x} : \texttt{any} \Rightarrow \emptyset \quad \texttt{\{integer,number\}} \vee^+ \texttt{any} = \texttt{\{integer,number\}}}{\emptyset;\Gamma \vdash^t \texttt{if b do \{i,n\} else x end} : \texttt{\{integer,number\}} \Rightarrow \emptyset}$$

$$\frac{\emptyset;\Gamma' \vdash^t \texttt{b} : \texttt{any} \Rightarrow \emptyset \quad \texttt{any} \prec^+ \texttt{boolean} \quad \emptyset;\Gamma' \vdash^t \texttt{\{i,n\}} : \texttt{\{integer,number\}} \Rightarrow \emptyset \quad \Delta;\Gamma' \vdash^t \texttt{x} : \texttt{\{number,any\}} \Rightarrow \emptyset \quad \texttt{\{integer,number\}} \vee^+ \texttt{\{number,any\}} = \texttt{\{number,number\}}}{\emptyset;\Gamma' \vdash^t \texttt{if b do \{i,n\} else x end} : \texttt{\{number,number\}} \Rightarrow \emptyset}$$

donde

$$\Gamma = [b \mapsto \texttt{any}, i \mapsto \texttt{integer}, n \mapsto \texttt{number}, x \mapsto \texttt{any}]$$

$$\Gamma' = [b \mapsto \texttt{any}, i \mapsto \texttt{integer}, n \mapsto \texttt{number}, x \mapsto \texttt{\{number,any\}}]$$

.

Aquí es posible observar que ya no se verifica el lema SGG (EXPR) -el más cercano a la formulación original de Siek et al. [2015] - ya que a partir de una mayor precisión en los tipos del contexto para el segundo ejemplo ($\Gamma' \lll \Gamma$), no se infiere un tipo más preciso que en el primero (no se cumple $\texttt{\{number,number\}} \lll \texttt{\{number,integer\}}$).

La hipótesis es, sin embargo, que en un sistema similar al actual pero basado en el subtipado-materialización ($\prec^+$), sí sería posible probar SGG, pero adaptando los lemas para usar $(\prec^+)^{op}$ en vez de $\lll$.

Capítulo 4

Elixir con casts

En el capítulo anterior se presentó un sistema de tipos estático para el fragmento de estudio, además de una extensión gradual basada en la noción de consistencia que permite integrar información dinámica a los chequeos en ese sistema. Una gradualización completa de Elixir con tipos en el sentido de la Gradual Guarantee, supone, además, de modificar el *runtime* para poder manejar la consistencia (e inconsistencia) en tiempo de ejecución.

Mientras que el sistema de Cassola et al. [2022] y el del capítulo anterior tienen valor en sí mismos -al establecer un sistema de chequeos que contempla programas parcialmente especificados-, toda la información adquirida durante el chequeo de un programa es ignorada en tiempo de ejecución, por lo que la evaluación ocurre sin diferencias respecto a la versión sin tipos. En contraste, el sistema que se propone en este capítulo define una traducción que inserta casts en posiciones críticas de los programas chequeados con el fin de evitar inconsistencias en runtime en los casos en que la información estática no es suficiente para asegurarlas.

En la primera sección se define la extensión del lenguaje con casts y la traducción a ese lenguaje en base al juicio $\vdash m \rightsquigarrow m'$, adaptando la técnica de Siek and Taha [2007] al contexto de este trabajo. En la segunda sección se formaliza la evaluación del lenguaje del fragmento al definir el juicio $\vdash^* m \Downarrow v$. La tercera sección extiende la definición formal de la evaluación para incluir programas que involucran casts. Considerando lo visto en las primeras tres secciones, se define la evaluación gradual $\vdash^* m \tilde{\Downarrow} v$ para los programas chequeados respecto al sistema del capítulo anterior: primero se reescribe el programa insertando los casts por consistencia y luego se evalúa siguiendo la definición anterior. Esto último se explica en la cuarta (y última) sección.

4.1. Traducción

Todos los objetos involucrados en el juicio de traducción $\Delta; \Gamma \vdash^t e \rightsquigarrow e' : \tau \Rightarrow \Gamma'$ al lenguaje con casts son de la misma forma que sus correspondientes en j. chequeo de expresión, sumado a que e' es una expresión que puede incluir casts de tipo. Como estos se representan por medio de un constructo sintáctico ajeno al fragmento, formalmente se debe recurrir a la definición del lenguaje extendido con casts para poder dar sentido completo al juicio de traducción. De aquí en adelante, el lenguaje original y el extendido con casts serán distinguidos bajo las denominaciones de *lenguaje fuente* y *lenguaje objeto* de la traducción respectivamente.

4.1.1. Sintaxis del lenguaje objeto

El lenguaje objeto luce exactamente idéntico al fuente, salvo por el hecho de que además se incluyen casts de tipo al nivel de las expresiones:

$$e \quad ::= \quad \ldots \mid (e \mid T \sim\!> T) \mid \ldots$$

Intuitivamente, en una expresión casteada $(e \mid \tau \sim\!> \sigma)$ el tipo τ que aparece en la izquierda representa al tipo previamente inferido - y por lo tanto "seguro"- para e; por el contrario, el tipo σ en la derecha representa al tipo decretado por medio del uso de la consistencia -y por lo tanto "inseguro"-. Como podía intuirse, el uso de la consistencia en el sistema de tipos compromete la correctitud respecto a la evaluación al permitir el reemplazo del tipo dinámico `any` por un tipo estático cualquiera. Por ejemplo, la siguiente inferencia válida

$$\frac{\emptyset; [x \mapsto \texttt{any}] \vdash^t \texttt{x} : \texttt{any} \Rightarrow \emptyset}{\emptyset; [x \mapsto \texttt{any}] \vdash^t \texttt{rem(x, 3)} : \texttt{integer} \Rightarrow \emptyset}$$

permite tratar a la variable `x` de tipo `any` como si fuese de tipo `integer` gracias al hecho que `any` $\sim$ `integer`. El problema es que en tiempo de ejecución el operador `rem` exige valores enteros para ambos argumentos; de lo contrario lanzará un `ArithmeticError`. Insertando un cast se puede remediar este hecho:

$$\frac{\emptyset; [x \mapsto \texttt{any}] \vdash^t \texttt{x} : \texttt{any} \Rightarrow \emptyset}{\emptyset; [x \mapsto \texttt{any}] \vdash^t \texttt{rem((x | any ~> integer), 3)} : \texttt{integer} \Rightarrow \emptyset}$$

El cast para el argumento de `rem` dentro de la expresión `rem((x | any ~> integer), 3)` debe resolver en tiempo de ejecución a `rem(x, 3)` cuando `x` es un entero y lanzar un `CastError` en cualquier otro caso. En este caso, la evaluación del cast aprovecha la información estática de `rem` para evitar lanzar un `ArithmeticError` por no poder realizar la operación con el argumento dado.

Tanto "~>" como "|" forman parte de un conjunto de operadores binarios infijos válidos para el parser de Elixir para ser usados en nuevas definiciones (veáse la sección *Operators* de la documentación oficial). Aquí en concreto, ambos se utilizan en conjunto para la definición de una macro del lenguaje que resuelve previo a la compilación, por lo que en realidad ninguno de los símbolos (ni los tipos) siquiera ocurren en el AST que se entrega como *input* para el compilador hacia BEAM. Este último punto se profundiza en el capítulo siguiente.

4.1.2. Sistema de tipos con casts explícitos

A pesar de que ambos tienen motivaciones diferentes, es posible trazar una analogía entre los casts por consistencia que se utilizan en el *gradual typing* y los casts por subtipado que se utilizan para definir la semántica de coerciones para lenguajes con subtipado (por ejemplo, en la Sección 15.6 de Pierce [2002]), en cuando ambos hacen la transición de un sistema con una conversión implícita de tipos (allí por subtipado, y aquí por consistencia) a otro que incluye una primitiva explícita que se inserta cada vez que se aplica la conversión. Tomando eso en cuenta, la regla T_CAST para el tipado de los casts viene sin sorpresas:

$$\frac{\Delta; \Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \tau \sim \sigma}{\Delta; \Gamma \vdash^t (e \mid \tau \sim\!> \sigma) : \sigma \Rightarrow \Gamma'} \text{(T\_CAST)}$$

La definición a alto nivel del sistema de tipos para el lenguaje objeto es la siguiente:

1. La sintaxis de los tipos es la de los tipos graduales ($\tilde{T}$).
2. Valen todas las reglas del sistema estático (el subtipado extendido con `any` como en la Sección 3.2.2.3).

3. Valen todas las reglas que simulan la especificación dinámica (T_DEF_ANY, T_ANON_ANY, T_DCALL_ANY).
4. Las reglas TP y TPR para el tipado de patrones se mantienen como en la versión gradual.
5. Se agrega la regla T_CAST que permite pasar del tipo inferido a cualquier tipo consistente siempre que se aplique el cast correspondiente.

En esencia, las reglas del sistema con casts explícitos extienden al sistema estático tratando al tipo `any` como a uno más de los tipos estáticos. Deben agregarse las reglas que simulan la especificación dinámica para declaraciones no especificadas, además de mantener las reglas graduales del chequeo de patrones. Con respecto a este último punto, en la Sección 6.2.2 se propone una alternativa que, entre otras cosas, permitiría un pasaje más natural al sistema de traducción para las reglas que involucran patrones.

La regla T_CAST permite recuperar toda la flexibilidad que otorga la consistencia en el sistema de tipos gradual, a pesar que aquí su aplicación debe ser justificada de manera explícita; y esto se hace insertando los casts correspondientes.

4.1.3. Operador de merge

En el sistema de tipos estático de Cassola et al. [2022], la regla T_SUB es útil en cuanto permite pasar implícitamente de un tipo τ a otro tipo σ para una expresión e siempre que se verifique $\tau \leq \sigma$. Debido a la naturaleza algorítmica del sistema en este trabajo, este hecho resulta menos obvio, aunque se justifica gracias al Teorema 16.2.5 de Pierce [2002], como se explicó anteriormente.

En el sistema gradual sucede un fenómeno similar con la consistencia: las reglas validan implícitamente el pasaje de τ a σ cuando $\tau \sim \sigma$. Los casts que se introducen en el sistema de tipos con casts explícitos hacen -valga la redundancia- explícito este hecho: cada aplicación de la consistencia se justifica con un cast. Por ejemplo, si `x` : `any` y `n` : `number`, en la expresión

```
if b do
  {x, 1}
else
  {n, 1.0}
end
```

deberían, a priori, insertarse casts en cada una de las ramas hacia `{number,number}`:

```
if b do
  ({x, 1} | {any, integer} ~> {number, number})
else
  ({n, 1.0} | {number, float} ~> {number, number})
end
```

Como se ve arriba, el fragmento del cast correspondiente a la primera coordenada hace explícita la aplicación de la consistencia en la primera (`any` $\sim$ `number`) y la segunda rama (`number` $\sim$ `number`). Por otro lado, el fragmento del cast correspondiente a la segunda coordenada de la tupla, se debe, en las dos ramas, al uso implícito del subtipado (`integer` $\leq$ `number` y `float` $\leq$ `number`). Para evitar que éste interfiera, los casts de ambas ramas se corrigen trivializando la segunda coordenada:

```
if b do
  ({x, 1} | {any, integer} ~> {number, integer})
else
```

```
    ({n, 1.0} | {number, float} ~> {number, float})
  end
```

En Siek and Taha [2007], la corrección del ejemplo anterior se plantea en términos más generales con la introducción del *operador de merge* (denotado con el símbolo "$\leftharpoonup$") entre dos tipos graduales cualesquiera. Dado un cast $(e \mid \tau \rightsquigarrow \sigma)$, el merge $\tau \leftharpoonup \sigma$ combina la información de τ y σ de modo tal de obtener un tipo consistente a τ y subtipo de σ. Por ejemplo:

- $\{\texttt{any},\texttt{integer}\} \leftharpoonup \{\texttt{number},\texttt{number}\} = \{\texttt{number},\texttt{integer}\}$
- $\{\texttt{number},\texttt{float}\} \leftharpoonup \{\texttt{number},\texttt{number}\} = \{\texttt{number},\texttt{float}\}$

Finalmente, cada cast $(e \mid \tau \rightsquigarrow \sigma)$ se reemplaza, por $(e \mid \tau \rightsquigarrow \tau \leftharpoonup \sigma)$. Siguiendo el ejemplo anterior:

- $(\{\texttt{x},1\} \mid \{\texttt{any},\texttt{integer}\} \rightsquigarrow \{\texttt{number},\texttt{number}\})$ se reemplaza por
  $(\{\texttt{x},1\} \mid \{\texttt{any},\texttt{integer}\} \rightsquigarrow \{\texttt{number},\texttt{integer}\})$
- $(\{\texttt{n},1.0\} \mid \{\texttt{number},\texttt{float}\} \rightsquigarrow \{\texttt{number},\texttt{number}\})$ se reemplaza por
  $(\{\texttt{n},1.0\} \mid \{\texttt{number},\texttt{float}\} \rightsquigarrow \{\texttt{number},\texttt{float}\})$

La definición del operador de merge es una reformulación de la original en Siek and Taha [2007], y se presenta en el siguiente procedimiento:

procedure MERGE(τ, σ)
  switch (τ, σ) **do**
    case (any, σ)
      σ
    case (τ, any)
      any
    case $([\tau], [\sigma])$
      return $[\text{MERGE}(\tau, \sigma)]$
    case $(\{\tau_1, \ldots, \tau_n\}, \{\sigma_1, \ldots, \sigma_n\})$
      return $\{\text{MERGE}(\tau_1, \sigma_1), \ldots, \text{MERGE}(\tau_n, \sigma_n)\}$
    case $(\%\{\overline{(k_i \Rightarrow \tau_i)}_{i \in I}\}, \%\{\overline{(l_j \Rightarrow \sigma_j)}_{i \in J}\})$ **when** $\text{SUBSET}(J, I)$
      $\%\{\overline{(k_i \Rightarrow \text{MERGE}(\tau_i, \sigma_i))}_{i \in I \cap J}, \overline{(k_i \Rightarrow \sigma_i)}_{i \in I \setminus J}\}$
    case $(\%\{\overline{(k_i \Rightarrow \tau_i)}_{i \in I}\}, \%\{\overline{(l_j \Rightarrow \sigma_j)}_{i \in J}\})$ **when** $\text{SUBSET}(I, J)$
      $\%\{\overline{(k_i \Rightarrow \text{MERGE}(\tau_i, \sigma_i))}_{i \in I}\}$
    case $((\tau_1, .., \tau_n) \to \tau_0, (\sigma_1, .., \sigma_n) \to \sigma_0)$
      return $(\text{MERGE}(\tau_1, \sigma_1), .., \text{MERGE}(\tau_n, \sigma_n)) \to \text{MERGE}(\tau_0, \sigma_0)$
    default
      τ
end procedure

Las propiedades básicas enunciadas en Siek and Taha [2007] se verifican directamente:

Propiedades básicas del operador de merge: Dados $\tau, \sigma \in \tilde{T}$, valen las siguientes propiedades:

1. $(\tau \leftharpoonup \tau) = \tau$
2. $\tau \sim (\tau \leftharpoonup \sigma)$
3. Si $\tau \lesssim \sigma$, entonces $(\tau \leftharpoonup \sigma) \leq \sigma$

Para ilustrar la Propiedad 3, se proponen algunos ejemplos:

$$(\texttt{\{integer,integer,any\}} \leftharpoonup \texttt{\{any,number,float\}}) = \texttt{\{any,integer,float\}}$$

$$(\texttt{\%\{:a => integer,:b => any\}} \leftharpoonup \texttt{\%\{:b => float\}}) = \texttt{\%\{:a => integer,:b => float\}}$$

$$(\texttt{\%\{:a => any\}} \to \{\}) \leftharpoonup (\texttt{\%\{:a => integer,:b => any\}} \to \{\}) = \texttt{\%\{:a => integer\}} \to \{\}$$

4.1.4. Juicios para la traducción

Para indicar que una expresión e del lenguaje fuente se traduce a otra expresión e' en el lenguaje objeto se define el juicio:

$$\Delta; \Gamma \vdash^t e \rightsquigarrow e' : \tau \Rightarrow \Gamma' \qquad \text{(j. traducción de expresión)}$$

Aquí $\tau \in \tilde{T}$ es un tipo de la sintaxis gradual, $\Delta \in \textsc{Map}(\textsc{Id} \times \mathbb{N}, \textsc{List}(\tilde{T}) \times \tilde{T})$ y $\Gamma, \Gamma' \in \textsc{Map}(\textsc{Id}, \tilde{T})$ como en la versión gradual de j. chequeo de expresión.

El juicio para la traducción indica simultáneamente tres cosas:

1. $\Delta; \Gamma \vdash^t e : \tau \Rightarrow \Gamma'$ es válido en el sistema de tipos gradual
2. e' es la traducción de e.

y a posteriori (esto se hace formal en la sección 4.1.7)

3. $\Delta; \Gamma \vdash^t e' : \tau \Rightarrow \Gamma'$ es válido en el sistema de tipos con casts explícitos

Además, a partir del juicio anterior se definen

$$\Delta \vdash^t d \rightsquigarrow d' \qquad \text{(j. traducción de declaración)}$$

$$\Delta \vdash^{ts} m \rightsquigarrow m' \qquad \text{(j. de traducción de programa bajo especificaciones)}$$

y finalmente también

$$\vdash^t m \rightsquigarrow m' \qquad \text{(j. traducción de programa)}$$

4.1.5. Reglas para la traducción de expresiones

Las reglas para la traducción de expresiones al lenguaje con casts se inspiran en Cimini and Siek [2016], aunque en este caso deben adaptarse para tomar en cuenta el subtipado. Afortunadamente, la incorporación del operador de merge resulta suficiente en la mayoría de los casos. Para los operadores básicos esto no basta y su traducción se define haciendo uso de la propiedad del Argumento máximo establecida en la Sección 3.1.8.3.1.

El sistema para la traducción está diseñado de tal forma que sus reglas están en relación de uno a uno con las reglas de tipado del sistema gradual: por cada regla T_[EXPR], se define la correspondiente TRANSLATE_[EXPR]. Además, el sistema para la traducción puede considerarse como un *enriquecimiento* del sistema de tipado gradual, ya que los juicios y reglas lucen exactamente iguales a menos de la nueva información correspondiente a la expresión traducida.

4.1.5.1. Reglas básicas

Las reglas que más directamente ilustran el espíritu de la traducción son TRANSLATE_CONS, TRANSLATE_IFE y TRANSLATE_CASE para los constructores de listas, if/else y definición por casos respectivamente. Para estos casos, la traducción se basa en la definición de la macro auxiliar $(e \mid \tau \sim\!\!\gg \sigma)$:

$$(e \mid \tau \sim\!\!\gg \sigma) = \begin{cases} e & \tau = (\tau \leftharpoonup \sigma) \\ (e \mid \tau \sim\!\!> (\tau \leftharpoonup \sigma)) & \text{si no} \end{cases} \qquad \text{(macro para el cast)}$$

que permite asegurar:

1. No se insertarán casts triviales de la forma $(e \mid \tau \sim\!\!> \tau)$
2. $\tau \sim \sigma$ cada vez que se inserte un cast $(e \mid \tau \sim\!\!> \sigma)$

Las tres reglas en cuestión son:

$$\frac{\Delta;\Gamma \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \vdash^t e_2 \rightsquigarrow e_2' : \tau_2 \Rightarrow \Gamma_2 \qquad [\tau_1] \widetilde{\vee} \tau_2 = \sigma}{\Delta;\Gamma \vdash^t [e_1 \mid e_2] \rightsquigarrow [(e_1' \mid \tau_1 \sim\!\!\gg \sigma) \mid (e_2' \mid \tau_2 \sim\!\!\gg [\sigma])] : [\sigma] \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (TRANSLATE\_CONS)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e \rightsquigarrow e' : \tau \Rightarrow \Gamma' \quad \tau \lesssim \texttt{boolean} \\ \Delta;\Gamma \dagger \Gamma' \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1 \quad \Delta;\Gamma \dagger \Gamma' \vdash^t e_2 \rightsquigarrow e_2' : \tau_2 \Rightarrow \Gamma_2 \quad \tau_1 \widetilde{\vee} \tau_2 = \sigma\end{array}}{\begin{array}{c}\Delta;\Gamma \vdash^t \texttt{if}\ e\ \texttt{do}\ e_1\ \texttt{else}\ e_2\ \texttt{end} \\ \rightsquigarrow \texttt{if}\ (e' \mid \tau \sim\!\!\gg \texttt{boolean})\ \texttt{do}\ (e_1' \mid \tau_1 \sim\!\!\gg \sigma)\ \texttt{else}\ (e_2' \mid \tau_2 \sim\!\!\gg \sigma)\ \texttt{end} : \sigma \Rightarrow \Gamma'\end{array}} \text{ (TRANSLATE\_IFE)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e \rightsquigarrow e' : \tau \Rightarrow \Gamma' \\ \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_1 \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1' \quad \cdots \quad \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_n \vdash^t e_n \rightsquigarrow e_n' : \tau_n \Rightarrow \Gamma_n' \\ \sigma = \widetilde{\bigvee}_{i=1}^{n} \tau_i\end{array}}{\begin{array}{c}\Delta;\Gamma \vdash^t \texttt{case}\ e\ \texttt{do}\ p_1\ \texttt{->}\ e_1\texttt{;}\ \ldots\texttt{;}\ p_n\ \texttt{->}\ e_n\ \texttt{end} \\ \rightsquigarrow \texttt{case}\ e'\ \texttt{do}\ p_1\ \texttt{->}\ (e_1' \mid \tau_1 \sim\!\!\gg \sigma)\texttt{;}\ \ldots\texttt{;}\ p_n\ \texttt{->}\ (e_n' \mid \tau_n \sim\!\!\gg \sigma)\ \texttt{end} : \sigma \Rightarrow \Gamma'\end{array}} \text{ (TRANSLATE\_CASE)}$$

Como ejemplo de aplicación de estas reglas, se puede ver que las siguiente expresiones compuestas

```
x, y, n, b # tipa como any, any, number, boolean
if b do
  [x | []]
  else
  [1 | [x | []]]
end

if y do
  [x | []]
  else
  [n | [x | []]]
end
```

se traducen a

```
x, n, y, b # tipa como any, number, boolean
if b do
  [x | []]
  else
  [(1 | integer ~> any) | [x | []]]
end

if (y | any ~> boolean) do
  ([x | []] | [any] ~> [number])
  else
  [n | ([x | []] | [any] ~> [number])]
end
```

En la primera expresión, las dos ramas tipan como `[any]`. La traducción correspondiente a la primera rama `([x | []] | [any] ~» [any])` se elimina por definición de ~». En la segunda rama, la traducción

`([(1 | integer ~» any)|([x|[]] | [any] ~» [any])] | [any] ~» [any])`

se simplifica a `[(1 | integer ~> any)|[x|[]]]` por el mismo motivo.

En la segunda expresión, la primera rama tipa como `[any]` y la segunda como `[number]`, por lo que el tipo resultado es `[any]` ∨ `[number]` = `[number]`. En la primera rama, el tipo de la lista es `[any]`, por lo que su cuerpo debe ser casteado hacia `[number]` para asegurar un resultado numérico en el caso `b = :true`. En la segunda rama, la traducción

`([(n | number ~» number)|([x|[]] | [any] ~» [number])] | [number] ~» [number])`

se simplifica a `[n|([x|[]] | [any] ~» [number])]`.

Finalmente, el operador de merge asegura que los casts siempre sean entre tipos consistentes. Por ejemplo

```
x, n, y, b # tipa como any, number, boolean
case n do
  1 -> ({1, x} | {integer, any} ~>> {number, number})
  2 -> ({x, 1.0} | {any, float} ~>> {number, number})
  3 -> ({1.0, 1} | {float, integer} ~>> {number, number})
end
```

se simplifica a

```
x, n, y, b # tipa como any, number, boolean
case n do
  1 -> ({1, x} | {integer, any} ~> {integer, number})
  2 -> ({x, 1.0} | {any, float} ~> {number, float})
  3 -> {1.0, 1}
end
```

4.1.5.2. Reglas para las llamadas a funciones

Las reglas TRANSLATE_ACALL, TRANSLATE_ACALL_ANY, TRANSLATE_DCALL y TRANSLATE_DCALL_ANY se ocupan de insertar casts para las dos versiones de llamadas a funciones que se existen en la sintaxis:

$$\frac{\Delta;\Gamma \vdash^t e \rightsquigarrow e' : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \Gamma_0 \quad \Delta;\Gamma \vdash^t e_1 \rightsquigarrow e'_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^t e_n \rightsquigarrow e'_n : \sigma_n \Rightarrow \Gamma_n \quad \sigma_1 \lesssim \tau_1 \ \cdots \ \sigma_n \lesssim \tau_n}{\Delta;\Gamma \vdash^t e.(e_1, \ldots, e_n) \rightsquigarrow e'.((e'_1 \mid \sigma_1 \sim\!\!\gg \tau_1), \ldots, (e'_n \mid \sigma_n \sim\!\!\gg \tau_n)) : \tau_0 \Rightarrow \Gamma_0\Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{(TRANSLATE\_ACALL)}$$

$$\frac{\Delta;\Gamma \vdash^t e \rightsquigarrow e' : \mathsf{any} \Rightarrow \Gamma_0 \quad \Delta;\Gamma \vdash^t e_1 \rightsquigarrow e'_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^t e_n \rightsquigarrow e'_n : \sigma_n \Rightarrow \Gamma_n \quad e'' = (e' \mid \mathsf{any} \sim\!\!> (\mathsf{any}, .., \mathsf{any}_n) \to \mathsf{any})}{\Delta;\Gamma \vdash^t e.(e_1, \ldots, e_1) \rightsquigarrow e''.((e'_1 \mid \sigma_1 \sim\!\!\gg \mathsf{any}), \ldots, (e'_n \mid \sigma_n \sim\!\!\gg \mathsf{any})) : \mathsf{any} \Rightarrow \Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{(TRANSLATE\_ACALL\_ANY)}$$

$$\frac{\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \quad \Delta;\Gamma \vdash^t e_1 \rightsquigarrow e'_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^t e_n \rightsquigarrow e'_n : \sigma_n \Rightarrow \Gamma_n \quad \sigma_1 \lesssim \tau_1 \ \cdots \ \sigma_n \lesssim \tau_n}{\Delta;\Gamma \vdash^t f\_name(e_1, \ldots, e_n) \rightsquigarrow f\_name((e_1 \mid \sigma_1 \sim\!\!\gg \tau_1), \ldots, (e_n \mid \sigma_n \sim\!\!\gg \tau_n)) : \tau_0 \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{(TRANSLATE\_DCALL)}$$

$$\frac{(f\_name, n) \notin \mathrm{dom}(\Delta) \quad \Delta;\Gamma \vdash^t e_1 \rightsquigarrow e'_1 : \tau_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^t e_n \rightsquigarrow e'_n : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^t f\_name(e_1, \ldots, e_n) \rightsquigarrow f\_name((e_1 \mid \tau_1 \sim\!\!\gg \mathsf{any}), \ldots, (e_n \mid \tau_n \sim\!\!\gg \mathsf{any})) : \mathsf{any} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{(TRANSLATE\_DCALL\_ANY)}$$

Las dos reglas para las llamadas anónimas corresponden, en conjunto, a la regla para la traducción de la aplicación de Siek and Taha [2007] (en la Figura 5). En ese artículo la distinción se realiza implícitamente en la definición de la función de matching $\triangleright$.

TRANSLATE_ACALL se ocupa de insertar casts alrededor de los argumentos $e_1, ..., e_n$ cuando la expresión invocada e es de un tipo funcional $(\tau_1, .., \tau_n) \to \tau_0$, partiendo en cada argumento e_i del tipo inferido σ_i y terminando en el tipo τ_i esperado para la posición i.

TRANSLATE_ACALL_ANY trata el caso en que la expresión e es de tipo dinámico any. Luego de reemplazar la hipótesis de tipo any por la de $(\mathsf{any}, .., \mathsf{any}_n) \to \mathsf{any}$ (siendo n la cantidad de argumentos aplicados), se insertan casteos en los argumentos tal como en TRANSLATE_ACALL. Además, se precisa insertar un cast extra alrededor de e para justificar el remplazo consistente de tipo any hacia $(\mathsf{any}, .., \mathsf{any}_n) \to \mathsf{any}$.

Finalmente, las reglas TRANSLATE_DCALL y TRANSLATE_DCALL_ANY para llamadas a funciones declaradas son análogas a las de funciones anónimas, salvo por el hecho de que no se precisa aplica ningún cast extra en el segundo caso.

Para ilustrar las reglas, se ofrece un ejemplo a partir del programa (correctamente tipado):

```
def Program do
  def untyped(x) do
    x
  end

  @spec inc(integer) :: integer
```

```
  def inc(x) do
      x + 1
  end

 @spec inc(any) :: any
  def unc(x) do
      x + 1
  end

  def main do
    x = untyped(:a)
    anonymous_inc = &inc/1
    anonymous_unc = &unc/1
    untyped_anonymous_inc = untyped(&inc/1)
    untyped_anonymous_unc = untyped(&unc/1)
    inc(x)
    unc(x)
    anonymous_inc.(x)
    anonymous_unc.(x)
    untyped_anonymous_inc.(x)
    untyped_anonymous_unc.(x)
  end
end
```

Para la porción del programa correspondiente al cuerpo de `main`, la traducción produce el resultado:

```
1   x = untyped(:a | :a ~> any)
2   anonymous_inc = &inc/1
3   anonymous_unc = &inc/1
4   untyped_anonymous_inc = untyped((&inc/1 | (integer -> integer) ~> any))
5   untyped_anonymous_unc = untyped((&unc/1 | (any -> any) ~> any))
6   inc(x | any ~> integer)
7   unc(x)
8   anonymous_inc.(x | any ~> integer)
9   anonymous_unc.(x)
10  (untyped_anonymous_inc | any ~> (any -> any)).(x)
11  (untyped_anonymous_unc | any ~> (any -> any)).(x)
```

A continuación se justifican las transformaciones:

- En la línea 5, aplica TRANSLATE_DCALL_ANY ya que `untyped` no está especificada:

$$\frac{(\texttt{untyped}, 1) \notin \Delta \qquad \Delta; \Gamma \vdash^{t} \texttt{\&inc/1} \rightsquigarrow \texttt{\&inc/1} : \texttt{integer} \to \texttt{integer} \Rightarrow []}{\Delta; \Gamma \vdash^{t} \texttt{untyped.(\&inc/1)} \rightsquigarrow \texttt{untyped.((\&inc/1 | integer} \to \texttt{integer ~> any))} : \texttt{any} \Rightarrow []}\ \text{(TRANSLATE\_DCALL\_ANY)}$$

  El cast de la línea 6 se explica de la misma manera.

- En la línea 7, aplica nuevamente TRANSLATE_DCALL:

$$\frac{\begin{array}{c}\Delta[\texttt{(inc, 1)}] = ((\texttt{integer}), \texttt{integer}) \\ \Delta; \Gamma \vdash^{t} \texttt{x} \rightsquigarrow \texttt{x} : \texttt{any} \Rightarrow [] \\ \texttt{any} \lesssim \texttt{integer}\end{array}}{\Delta; \Gamma \vdash^{t} \texttt{inc.(x)} \rightsquigarrow \texttt{inc.((x | any ~> integer))} : \texttt{any} \Rightarrow []} \text{(TRANSLATE\_DCALL)}$$

En la línea 8 la derivación es la misma pero la traducción se simplifica ya que `integer` se reemplaza por `any` y el cast de `any` a `any` se omite -de acuerdo a la definición de macro para el cast-.

- En la línea 9, aplica TRANSLATE_ACALL ya que `anonymous_inc` es una expresión de tipo funcional (`integer` → `integer`)

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash^{t} \texttt{anonymous\_inc} \rightsquigarrow \texttt{anonymous\_inc} : \texttt{integer} \to \texttt{integer} \Rightarrow \emptyset \\ \Delta; \Gamma \vdash^{t} \texttt{x} \rightsquigarrow \texttt{x} : \texttt{any} \Rightarrow [] \\ \texttt{any} \lesssim \texttt{integer}\end{array}}{\Delta; \Gamma \vdash^{t} \texttt{anonymous\_inc.(x)} \rightsquigarrow \texttt{inc.((x | any ~> integer))} : \texttt{any} \Rightarrow []} \text{(TRANSLATE\_ACALL)}$$

En la línea 10 la traducción se trivializa por el mismo motivo que el de la línea 8.

- Para las últimas dos lineas aplica directamente la regla TRANSLATE_ACALL_ANY porque el tipo de la variable `untyped_anonymous_inc` es `any`.

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash^{t} \texttt{untyped\_anonymous\_inc} \rightsquigarrow \texttt{untyped\_anonymous\_inc} : \texttt{any} \Rightarrow \emptyset \\ \Delta; \Gamma \vdash^{t} \texttt{x} \rightsquigarrow \texttt{x} : \texttt{any} \Rightarrow []\end{array}}{\begin{array}{c}\Delta; \Gamma \vdash^{t} \texttt{untyped\_anonymous\_inc.(x)} \\ \rightsquigarrow \texttt{(untyped\_anonymous\_inc | any} \to \texttt{any ~> any).(x)} : \texttt{any} \Rightarrow []\end{array}} \text{(TRANSLATE\_ACALL\_ANY)}$$

Notar que nuevamente el cast del argumento se trivializa ya que ambos tipos son `any`.

Para el ejemplo anterior, en caso de intercambiar primera la línea del cuerpo de `main` por `x = 1`, el resultado de la traducción se vuelve:

```
x = 1
anonymous_inc = &inc/1
anonymous_unc = &unc/1
untyped_anonymous_inc = untyped((&inc/1) | (integer -> integer) ~> any)
untyped_anonymous_unc = untyped((&unc/1) | (any -> any) ~> any)
inc(x)
unc(x | integer ~> any)
anonymous_inc.(x)
anonymous_unc.(x | integer ~> any)
(untyped_anonymous_inc | any ~> (any -> any)).(x | integer ~> any)
(untyped_anonymous_unc | any ~> (any -> any)).(x | integer ~> any)
```

Las justificaciones son similares al caso anterior.

4.1.5.3. Reglas triviales

A excepción de las reglas para operadores, las restantes reglas del sistema de tipos gradual se reflejan en reglas de traducción “triviales”, en el siguiente sentido:

1. Si en las hipótesis no ocurre ningún juicio de tipado para alguna subexpresión, la regla de traducción en cuestión corresponde a un caso base y la expresión se traduce a sí misma (es decir, sin cambios).

2. Por cada otra regla de tipado, la regla de traducción surge de reemplazar cada hipótesis del juicio de tipado por el juicio de traducción respectivo y aplicar estructuralmente los resultados para obtener la expresión traducida.

El primero es el caso de TRANSLATE_LIT, TRANSLATE_VAR, TRANSLATE_ELIST y TRANSLATE_ANON. Por ejemplo:

$$\frac{\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0)}{\Delta; \Gamma \vdash^t \&f\_name/n \rightsquigarrow \&f\_name/n : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \emptyset} \text{ (TRANSLATE\_ANON)}$$

El segundo caso comprende las reglas TRANSLATE_MATCH, TRANSLATE_TUPLE, TRANSLATE_MAP y TRANSLATE_SEQ. Por ejemplo:

$$\frac{\Delta; \Gamma \vdash^t e \rightsquigarrow e' : \tau \Rightarrow \Gamma' \qquad \Gamma; \tau \vdash^{pt} p : \sigma \Rightarrow \Gamma'}{\Delta; \Gamma \vdash^t p \ = \ e \rightsquigarrow p \ = \ e' : \sigma \Rightarrow \Gamma' \dagger \Gamma''} \text{ (TRANSLATE\_MATCH)}$$

$$\frac{\Delta; \Gamma \vdash^t e_1 \rightsquigarrow e'_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \cdots \qquad \Delta; \Gamma \vdash^t e_n \rightsquigarrow e'_n : \tau_n \Rightarrow \Gamma_n}{\Delta; \Gamma \vdash^t \{e_1, \ldots, e_n\} \rightsquigarrow \{e'_1, \ldots, e'_n\} : \{\tau_1, \ldots, \tau_n\} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (TRANSLATE\_TUPLE)}$$

En el Anexo D.2 se encuentra una presentación exhaustiva de todas éstas reglas.

4.1.5.4. Reglas para operadores

La dificultad para definir la traducción de las operaciones básicas respecto al sistema de tipos del capítulo anterior ya se puede motivar con el siguiente ejemplo sencillo:

```
@spec untyped_sum(any, any) :: any
def untyped_sum(x, y) do
  x + y
end
```

Para un primer intento se podría tomar la opción de decretar que los argumentos serán enteros, y en ese caso reescribir a

```
@spec untyped_sum(any, any) :: any
def untyped_sum(x, y) do
  (x | any ~> integer) + (y | any ~> integer)
end
```

Se ve fácilmente que esta opción muestra algunos escenarios de funcionamiento correcto y otro de funcionamiento incorrecto:

1. Primer escenario correcto: cuando `x` e `y` resultan ser de tipo integer, los casts en tiempo de ejecución serán exitosos y la suma se ejecutará normalmente.

2. Segundo escenario correcto: cuando o bien `x` o bien `y` resultan ser tipos donde la suma no tiene sentido (por ejemplo `atom`), alguno de los casts resultará en un `CastError` previo a la evaluación, sin llegar a un `ArithmeticError` por argumentos incorrectos para la suma.

3. Escenario incorrecto: cuando `x` e `y` son numéricos pero alguno de ellos corresponde a un flotante, la evaluación resulta en un `CastError` como en el caso anterior a pesar de que la suma hubiera tenido sentido (y tampoco hubiera habido error de tipo en caso de haber precisado los tipos en la especificación).

En caso de tomar la opción de castear los argumentos a `float` en lugar de `integer`, es claro que se obtienen beneficios y defectos análogos.

Como tercera alternativa, se plantea castear los argumentos a `number`, el tipo que contiene a todos los numéricos (aquí valores de tipo `integer` y `float`).

```
@spec untyped_sum(any, any) :: any
def untyped_sum(x, y) do
  (x | any ~> number) + (y | any ~> number)
end
```

Ahora se observa que el escenario descrito en el punto 3 pierde vigencia, ya que tanto valores enteros como flotantes serán exitosos en el cast hacia `number`. Por otro lado, el `CastError` se mantiene para cualquier otro valor.

En general, `number` funciona mientras que `integer` y `float` no, ya que `number` es el tipo que contiene a todos los valores para los cuales la suma tiene sentido. Formalmente, este hecho se puede enunciar de la siguiente manera:

$$\begin{gathered} \mathit{sumType}(\texttt{number}, \texttt{number}) \text{ está definido y para cualquier otro par } \tau_1, \tau_2 \in T \\ \text{tal que } \mathit{sumType}(\tau_1, \tau_2) \text{ está definido, } \tau_1 \leq \texttt{number} \text{ y } \tau_2 \leq \texttt{number} \end{gathered}$$

La propiedad del Argumento máximo para operadores de la Sección 3.1.8.3.1 generaliza el enunciado anterior para cualquiera de los operadores binarios $\oplus$ del fragmento con respecto a la función $\mathit{binOpType}$:

$$\begin{gathered} \text{Para cada } \oplus, \text{ existen } \tau_1^{\oplus}, \tau_2^{\oplus} \in T \text{ tales que} \\ \mathit{binOpType}(\oplus, \tau_1^{\oplus}, \tau_2^{\oplus}) \text{ está definido y para cualquier otro par } \tau_1, \tau_2 \in T \\ \text{tal que } \mathit{binOpType}(\oplus, \tau_1, \tau_2) \text{ está definido, } \tau_1 \leq \tau_1^{\oplus} \text{ y } \tau_2 \leq \tau_2^{\oplus} \end{gathered}$$

Lo razonable es, entonces, utilizar a los tipos $\tau_1^{\oplus}$ y $\tau_2^{\oplus}$ para el cast de los argumentos. La regla TRANSLATE_BIN_OP queda expresada como:

$$\frac{\begin{gathered} \Delta; \Gamma \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta; \Gamma \vdash^t e_2 \rightsquigarrow e_2' : \tau_2 \Rightarrow \Gamma_2 \\ \tau_1 \leftharpoonup \tau_1^{\oplus} = \tau_1' \quad \tau_2 \leftharpoonup \tau_2^{\oplus} = \tau_2' \\ \widetilde{\mathit{binOpType}}(\oplus, \tau_1, \tau_2) = \sigma \quad \mathit{binOpType}(\oplus, \tau_1', \tau_2') = \sigma' \end{gathered}}{\Delta; \Gamma \vdash^t e_1 \oplus e_2 \rightsquigarrow ((e_1' \mid \tau_1 \sim\!\!\gg \tau_1') \oplus (e_2' \mid \tau_2 \sim\!\!\gg \tau_2') \mid \sigma' \sim\!\!\gg \sigma) : \sigma \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (TRANSLATE\_BIN\_OP)}$$

De manera análoga, se define la regla para operadores unarios:

$$\frac{\begin{gathered} \Delta; \Gamma \vdash^t e \rightsquigarrow e' : \tau \Rightarrow \Gamma_1 \quad \tau \leftharpoonup \tau^{\ominus} = \tau' \\ \widetilde{\mathit{uniOpType}}(\ominus, \tau) = \sigma \quad \mathit{uniOpType}(\ominus, \tau') = \sigma' \end{gathered}}{\Delta; \Gamma \vdash^t \ominus e \rightsquigarrow (\ominus (e_1' \mid \tau \sim\!\!> \tau') \mid \sigma' \sim\!\!> \sigma) : \sigma \Rightarrow \Gamma_1} \text{ (TRANSLATE\_UNI\_OP)}$$

4.1.6. Reglas para la traducción de un programa

La traducción de un programa se define a partir de la única regla TRANSLATE_PROG, que es un reflejo de la regla T_PROG del sistema de tipos:

$$\frac{\vdash^{c} m \Rightarrow \Delta \qquad \Delta \vdash^{ts} m \rightsquigarrow m'}{\vdash^{t} m \rightsquigarrow m'} \text{ (TRANSLATE\_PROG)}$$

donde la traducción bajo un contexto de especificaciones Δ se define iterando la traducción de todas las declaraciones:

$$\frac{\Delta \vdash^{t} d_1 \rightsquigarrow d_1' \quad \ldots \quad \Delta \vdash^{t} d_n \rightsquigarrow d_n'}{\begin{array}{c}\Delta \vdash^{t} \texttt{defmodule Program do } d_1\texttt{, ..., } d_n \texttt{ end} \\ \rightsquigarrow \texttt{defmodule Program do } d_1'\texttt{, ..., } d_n' \texttt{ end}\end{array}} \text{ (TRANSLATE\_PROG\_AUX)}$$

y la traducción de las declaraciones se define a partir de las tres reglas:

$$\frac{}{\Delta \vdash s \rightsquigarrow s} \text{ (TRANSLATE\_SPEC)}$$

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \emptyset; \emptyset; \tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset; \Gamma_{n-1}; \tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta; \Gamma_n \vdash^{t} e : \sigma \Rightarrow \Gamma' \quad \sigma \lesssim \tau_0\end{array}}{\begin{array}{c}\Delta \vdash^{t} \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end} \\ \rightsquigarrow \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do (}e \texttt{ | } \sigma \sim\!\!\!\gg \tau_0\texttt{) end}\end{array}} \text{ (TRANSLATE\_DEF)}$$

$$\frac{\begin{array}{c}(f\_name, n) \notin \mathrm{dom}(\Delta) \\ \emptyset; \emptyset; \texttt{any} \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset; \Gamma_{n-1}; \texttt{any} \vdash^{pc} p_n \Rightarrow \Gamma_n\end{array}}{\begin{array}{c}\Delta \vdash^{t} \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end} \\ \rightsquigarrow \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do (}e \texttt{ | } \sigma \sim\!\!\!\gg \texttt{any) end}\end{array}} \text{ (TRANSLATE\_DEF\_ANY)}$$

El cast alrededor del cuerpo de la declaración en TRANSLATE_DEF se necesita para garantizar que el valor retornado tenga el tipo τ_0 decretado por la especificación, siendo que las regla T_DEF únicamente exigen que se infiera un tipo $\sigma \lesssim \tau_0$. En TRANSLATE_DEF_ANY el cast del cuerpo se explica de forma similar.

4.1.7. Correctitud respecto al sistema de tipos

El objetivo de esta sección es justificar la correctitud de la traducción respecto al sistema de tipos; otorgando un bosquejo de la demostración de los siguientes dos hechos:

1. Cualquier programa válido en el sistema gradual se traduce exitosamente
2. El resultado de la traducción de un programa válido es también un programa válido (en el sistema con casts explícitos).

Formalmente, los dos puntos anteriores se expresan en el siguiente enunciado:

Correctitud estática de la traducción: Si $\vdash^{t} m$ es válido en el sistema gradual, entonces existe un único m' en el lenguaje objeto tal que $\vdash^{t} m \rightsquigarrow m'$ y $\vdash^{t} m'$ respecto al sistema con casts explícitos.

La prueba se apoya fuertemente en el hecho que las reglas para la traducción de expresiones están en relación de uno a uno con las del sistema gradual. Al igual que para la prueba de la *Static Gradual Guarantee*, se precisan varios lemas auxiliares para los distintos juicios. A continuación enunciamos el más importante:

Correctitud estática de la traducción (EXPR): Si $\Delta;\Gamma \vdash^t e : \sigma \Rightarrow \Gamma$ se verifica en el sistema gradual, entonces existe un único e' tal que $\Delta;\Gamma \vdash^t e \rightsquigarrow e' : \sigma \Rightarrow \Gamma$ y $\Delta;\Gamma \vdash^t e' : \sigma \Rightarrow \Gamma$ se verifica respecto al sistema con casts explícitos.

La prueba del lema anterior se basa en un argumento inductivo respecto a $\Delta;\Gamma \vdash^t e : \sigma \Rightarrow \Gamma$ que permite construir la derivación de $\Delta;\Gamma \vdash^t e' : \sigma \Rightarrow \Gamma$ y la mayoría de los casos son directos. El caso más interesante es el que involucra a los operadores básicos; para éste se asume que la última regla aplicada fue T_UNI_OP (T_BIN_OP):

1. Por inversión T_UNI_OP, se sabe que existen e_0, τ de modo que:

   a) $e = \ominus e_0$

   b) $\Delta;\Gamma \vdash^t e_0 : \tau \Rightarrow \Gamma'$

   c) $\widetilde{uniOpType}(\tau) = \sigma$

2. Aplicando inductivamente el lema para e_0 se deduce $\Delta;\Gamma \vdash^t e_0 \rightsquigarrow e'_0 : \tau \Rightarrow \Gamma'$ para cierto e'_0 tal que $\Delta;\Gamma \vdash^t e'_0 : \tau \Rightarrow \Gamma$.

3. Como $\tau \in B \cup \{\mathsf{any}\}$ y $\tau^{\ominus} \in B$, se tiene por definición del operador de merge:

$$\tau' = \tau \leftharpoonup \tau^{\ominus} = \begin{cases} \tau, & \text{si } \tau \in B \\ \tau^{\ominus}, & \text{si } \tau = \mathsf{any} \end{cases}$$

4. Del punto anterior se deduce que $uniOpType(\ominus, \tau')$ está definido:

   - Si $\tau' = \tau$, $uniOpType(\ominus, \tau') = uniOpType(\ominus, \tau) = \widetilde{uniOpType}(\ominus, \tau) = \sigma$, donde la segunda igualdad se justifica por la conservatividad del lifting AGT, teniendo en cuenta que $\tau \in B$.
   - Si $\tau' = \tau^{\ominus}$, entonces por definición de $\tau^{\ominus}$ se deduce que $uniOpType(\ominus, \tau')$ está definido.

   Sea $uniOpType(\ominus, \tau') = \sigma'$

5. De 1., 2. y 4. se deduce por aplicación directa de TRANSLATE_UNI_OP:

$$\frac{\Delta;\Gamma \vdash^t e_0 \rightsquigarrow e'_0 : \tau \Rightarrow \Gamma' \quad \tau \leftharpoonup \tau^{\ominus} = \tau' \qquad \widetilde{uniOpType}(\ominus, \tau) = \sigma \quad uniOpType(\ominus, \tau') = \sigma'}{\Delta;\Gamma \vdash^t \ominus e_0 \rightsquigarrow (\ominus (e'_0 \mid \tau \sim\!> \tau') \mid \sigma' \sim\!> \sigma) : \sigma \Rightarrow \Gamma'} \text{ (TRANSLATE\_UNI\_OP)}$$

6. Por la propiedad 2. del operador de merge se sabe que $\tau \sim (\tau \leftharpoonup \tau^{\ominus}) = \tau'$.

7. Discutiendo por casos como en 3. resulta claro que $\tau \lll (\tau \leftharpoonup \tau^{\ominus}) = \tau'$. Luego, por monotonía de $\widetilde{uniOpType}$ respecto a $\lll$ se deduce $\sigma' \lll \sigma$, y en particular, $\sigma \sim \sigma'$.

8. La siguiente derivación en el sistema con casts explícitos:

$$\dfrac{\dfrac{\dfrac{\Delta;\Gamma \vdash^t e_0' : \tau \Rightarrow \Gamma' \qquad \tau \sim \tau'}{\Delta;\Gamma \vdash^t (e_0' \mid \tau \sim\!> \tau') : \tau' \Rightarrow \Gamma'}\,\text{(T\_CAST)} \qquad \mathit{uniOpType}(\tau') = \sigma'}{\Delta;\Gamma \vdash^t \ominus(e_0' \mid \tau \sim\!> \tau') : \sigma' \Rightarrow \Gamma'}\,\text{(T\_UNI\_OP)} \qquad \sigma \sim \sigma'}{\Delta;\Gamma \vdash^t (\ominus (e_0' \mid \tau \sim\!> \tau') \mid \sigma' \sim\!> \sigma) : \sigma \Rightarrow \Gamma'}\,\text{(T\_CAST)}$$

permite validar el juicio $\Delta;\Gamma \vdash^t \ominus e_0 \rightsquigarrow (\ominus (e_0' \mid \tau \sim\!> \tau') \mid \sigma' \sim\!> \sigma) : \tau \Rightarrow \Gamma'$ como se requería.

4.2. Evaluación en el lenguaje fuente

A continuación se propone un modelo de evaluación para el lenguaje fuente de la traducción (es decir, el fragmento de Elixir considerado en los dos capítulos anteriores) basado en la definición de una semántica de tipo *big-step*.

Es importante enfatizar el hecho de que el modelo que se presenta no pretende capturar los conceptos relevantes a la compilación hacia bytecode ni a la ejecución de la máquina virtual BEAM; únicamente se interesa en modelar los aspectos de la ejecución del lenguaje que permiten obtener una representación correcta de la semántica de evaluación.

Las desventajas de utilizar una definición *big-step* para la semántica son conocidas (ver Sección 7.2 y 7.3 de Harper [2016], por ejemplo) y en particular impiden sentar las bases para un análisis de *type safety* y de establecer una prueba para la *Dynamic Gradual Guarantee* (pues normalmente se aplica inducción en una semántica *small-step*, como en Siek et al. [2015]). El desarrollo de una semántica *small-step*, requiere -en este caso-, una elaboración mayor a lo usual, dadas las particularidades del lenguaje en cuanto respecta a la definición del alcance de los contextos en varias de las construcciones del lenguaje. Algunas de las dificultades para lograr una formalización de esta semántica se detallan en 6.2.5

La sección está organizada en tres partes:

1. Se define el conjunto de resultados (r) de la evaluación, formado por los valores (v) del lenguaje (definidos como una subsintaxis de las expresiones) y los errores (Err) de ejecución.
2. Se presenta la relación de evaluación $\Downarrow$ que hace explícita la reducción de expresiones a resultados.
3. Se define la evaluación para un programa que provea una declaración de `main/0`.

4.2.1. Resultados de la evaluación

Como es estándar en la definición de una semántica *big-step*, los valores representan expresiones completamente evaluadas y los errores corresponden a excepciones respecto a cierto requerimiento en el proceso de evaluación.

$$\begin{array}{rcl} v & ::= & l \mid [\,] \mid [v \mid v] \mid \{v, \ldots, v\} \mid \%\{l \Rightarrow v, \ldots, l \Rightarrow v\} \mid \&f\_name/arity \\ TypeErr & ::= & \texttt{BadBooleanError} \mid \texttt{ArithmeticError} \mid \texttt{BadFunctionError} \mid \texttt{BadArityError} \\ Err & ::= & TypeErr \mid \texttt{MatchError} \mid \texttt{CaseClauseError} \mid \texttt{FunctionClauseError} \\ r & ::= & v \mid Err \end{array}$$

La sintaxis de Err toma en cuenta únicamente a aquellos errores de Elixir que se pueden encontrar dentro de la evaluación del fragmento, y se omite el resto. La lista completa con todos los errores posibles del lenguaje se encuentra en el archivo `exceptions.ex`[1] del repositorio oficial.

[1] ver https://github.com/elixir-lang/elixir/blob/v1.14/lib/elixir/lib/exception.ex

Entre los errores de la sintaxis de Err se distingue la subsintaxis $TypeErr$ que incluye los casos que, en una eventual prueba de *type safety*, deberían poder objetarse como (casi[2]) imposibles en el contexto de un programa chequeado en el sistema estático. El resto de los errores comprende los casos que, al menos para el sistema actual, serían totalmente factibles en la ejecución de programas chequeados.

El resultado r de una evaluación correcta (sin errores) tomará la forma de un valor; por el contrario, de haber una excepción el error particular indica el motivo de la falla.

4.2.2. Reglas para la evaluación expresiones

4.2.2.1. Juicio de la evaluación

El juicio que indica la evaluación de una expresión e para obtener un resultado r se expresa como

$$\Delta; \Gamma \vdash^* e \Downarrow r \Rightarrow \Gamma' \qquad \text{(j. evaluación de expresión)}$$

Aquí Γ representa el contexto de valores previo a la evaluación, a tomar en cuenta para la sustitución de variables, y por lo tanto, es de la forma $\textsc{Map}(\textsc{Id}, V)$. Γ' es de la misma forma que Γ y corresponde al contexto exportado por la evaluación: contiene la información de todas las ligaduras que se generan al evaluar la expresión e. Finalmente, en el contexto $\Delta \in \textsc{Map}(\textsc{Id} \times \mathbb{N}, \textsc{List}(\textsc{List}(P) \times E))$ hace corresponder a cada (f_name, n) con los pares formados por la lista de n parámetros (P) y el cuerpo (E) para todas las declaraciones de f_name con aridad n.

4.2.2.2. Reglas sin errores

A continuación se presentan las reglas que definen la evaluación para los casos que ocurren sin errores; es decir, donde el juicio de evaluación para la conclusión toma la forma particular

$$\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma'$$

Los casos E_LIT y E_VAR se ocupan de los escenarios donde la expresión a evaluar es un literal y una variable respectivamente.

$$\frac{}{\Delta; \Gamma \vdash^* l \Downarrow l \Rightarrow \emptyset}\ \text{(E\_LIT)} \qquad \frac{\Gamma[x] = v}{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \emptyset}\ \text{(E\_VAR)}$$

Notar que, en base a la definición del juicio, en E_LIT el resultado v coincide con la expresión e y el contexto exportado es vacío; esto es propio de la evaluación de los valores. Por el contrario, en E_VAR la variable x se reduce al valor v que ocupa su identificador en el contexto.

Para la definición de E_UNI_OP y E_BIN_OP, se asumen definidas las funciones parciales $uniOpVal$ y $binOpVal$ que asignan el resultado de la operaciones $\ominus$ y $\oplus$ literal a partir de los argumentos literales (cuando éste esté definido). La definición de estas funciones ha quedado fuera del alcance del trabajo, pero deberían poder formalizarse sin mucho esfuerzo a partir de los ejemplos guiados de la Sección 2.5.3.

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \qquad uniOpVal(\ominus, v) = v'}{\Delta; \Gamma \vdash^* \ominus e \Downarrow v' \Rightarrow \Gamma'}\ \text{(E\_UNI\_OP)}$$

[2]Deben admitirse algunas excepciones como el `ArithmeticError` para la división entre 0. Para capturar las excepciones dentro de la prueba, puede utilizarse la técnica de blame tracking

$$\frac{\Delta;\Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \vdash^* e_2 \Downarrow v_2 \Rightarrow \Gamma_2 \qquad binOpVal(\oplus, v_1, v_2) = v_3}{\Delta;\Gamma \vdash^* e_1 \oplus e_2 \Downarrow v_3 \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (E\_BIN\_OP)}$$

Para el *pattern matching*, la regla de evaluación es precedida por la definición de la función parcial auxiliar $match(\Gamma, \Delta, p, v)$ que en caso de estar definida, devuelve las variables ligadas dentro del patrón p en un nuevo contexto de valores Γ'. La implementación de la función *match* se puede ver en el Recuadro 4.1.

$$\frac{\Delta;\Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \qquad match(\Gamma, \emptyset, p, v) = \Gamma''}{\Delta;\Gamma \vdash^* p \ = \ e \Downarrow v \Rightarrow \Gamma' \dagger \Gamma''} \text{ (E\_MATCH)}$$

```
procedure MATCH(Σ, Γ, p, v)
    switch (p, v) do
        case (l, l)
            return ∅
        case (&f_name/n, &f_name/n)
            return ∅
        case ([ ], [ ])
            return ∅
        case (x, v) when x ∉ Γ or Γ[x] = v
            return Γ[x ↦ v]
        case (^x, v) when Σ[x] = v
            return Γ
        case ([p1 | p2], [v1 | v2])
            Γ ← MATCH(Σ, Γ, p1, v1)
            return MATCH(Σ, Γ, p2, v2)
        case ({p1 , ... , pn}, {v1 , ... , vn})
            for i ∈ [1, ..., n] do
                Γ ← MATCH(Σ, Γ, pi, vi)
            end for
            return Γ
        case (%{(k̄i => pi)i∈I}, %{(k̄i => vi)i∈J}) when SUBSET(I, J)
            for i ∈ I do
                Γ ← MATCH(Σ, Γ, pi, vi)
            end for
            return Γ
        default
            fail
end procedure
```

Recuadro 4.1: Definición del algoritmo de match

Los casos de estructuras de datos son simples y pueden inferirse por analogía a las reglas de tipado correspondientes:

$$\frac{}{\Delta;\Gamma \vdash^* [\,] \Downarrow [\,] \Rightarrow \emptyset} \text{ (E\_ELIST)}$$

$$\frac{\Delta;\Gamma \vdash^* e_i \Downarrow v_i \Rightarrow \Gamma_i \quad i = 1, 2}{\Delta;\Gamma \vdash^* [e_1 \,|\, e_2] \Downarrow [v_1 \,|\, v_2] \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (E\_CONS)}$$

$$\frac{\Delta;\Gamma \vdash^* e_1 \Downarrow \tau_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^* e_n \Downarrow \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^* \{e_1,...,e_n\} \Downarrow \{v_1,...,v_n\} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (E\_TUPLE)}$$

$$\frac{\Delta;\Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^* e_n \Downarrow v_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^* \texttt{\%\{}(\overline{k_i \texttt{ => } e_i})_{i\in[n]}\texttt{\}} \Downarrow \texttt{\%\{}(\overline{k_i \texttt{ => } v_i})_{i\in[n]}\texttt{\}} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (E\_MAP)}$$

Para ilustrar el uso de las reglas anteriores, a continuación se describe la derivación que corresponde al juicio de evaluación sin errores:

$$\emptyset;\emptyset \vdash^* \texttt{\{y,y\} = \{x = 2,(y = 1)+1\}} \Downarrow \texttt{\{2,2\}} \Rightarrow [x \mapsto 2, y \mapsto 1]$$

La última regla aplicada es la siguiente instancia de E_MATCH:

$$\frac{\emptyset;\emptyset \vdash^* \texttt{\{x = 2,(y = 1)+1\}} \Downarrow \texttt{\{2,2\}} \Rightarrow [x \mapsto 2, y \mapsto 1] \qquad \mathit{match}(\emptyset, \emptyset, \texttt{\{y,y\}}, \texttt{\{2,2\}}) = [y \mapsto 2]}{\emptyset;\emptyset \vdash^* \texttt{\{y,y\} = \{x = 2,(y = 1)+1\}} \Downarrow \texttt{\{2,2\}} \Rightarrow [x \mapsto 2, y \mapsto 1]}$$

En este caso, se comprueba exitosamente que el valor `{2,2}` resultado de la evaluación de la expresión de la izquierda cumple con el patrón `{y,y}` de la derecha, y liga la variable `y` al valor `2` (sobreescribiendo la ligadura anterior).

La hipótesis de la izquierda para la regla anterior se valida por aplicación de E_TUPLE:

$$\frac{\emptyset;\emptyset \vdash^* \texttt{x = 2} \Downarrow \texttt{2} \Rightarrow [x \mapsto 2] \qquad \emptyset;\emptyset \vdash^* \texttt{(y = 1)+1} \Downarrow \texttt{2} \Rightarrow [y \mapsto 1]}{\emptyset;\emptyset \vdash^* \texttt{\{x = 2,(y = 1)+1\}} \Downarrow \texttt{\{2,2\}} \Rightarrow [x \mapsto 2, y \mapsto 1]}$$

que ubica los valores resultados de la evaluación en cada una de las coordenadas en una tupla, además de combinar los contextos para las variables exportadas. Finalmente, las derivaciones para las dos hipótesis en esta regla son:

$$\frac{\emptyset;\emptyset \vdash^* \texttt{2} \Downarrow \texttt{2} \Rightarrow \emptyset \qquad \mathit{match}(\emptyset, \emptyset, \texttt{x}, 2) = [x \mapsto 2]}{\emptyset;\emptyset \vdash^* \texttt{x = 2} \Downarrow \texttt{2} \Rightarrow [x \mapsto 2]}$$

$$\frac{\dfrac{\emptyset;\emptyset \vdash^* \texttt{1} \Downarrow \texttt{1} \Rightarrow \emptyset \qquad \mathit{match}(\emptyset, \emptyset, \texttt{y}, 1) = [y \mapsto 1]}{\emptyset;\emptyset \vdash^* \texttt{y = 1} \Downarrow \texttt{1} \Rightarrow [y \mapsto 1]} \qquad \emptyset;\emptyset \vdash^* \texttt{1} \Downarrow \texttt{1} \Rightarrow \emptyset \qquad \mathit{binOpValue}(+, 1, 1) = 2}{\emptyset;\emptyset \vdash^* \texttt{(y = 1)+1} \Downarrow \texttt{2} \Rightarrow [y \mapsto 1]}$$

Cuando la expresión es una definición por casos, la regla para la evaluación correcta (es decir, asumiendo que aplica una de las ramas y su resultado es correcto) es:

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \\ \mathit{match}(\Gamma', \emptyset, p_i, e_i) = \textbf{fail} \quad \forall i \in [1,...,k-1] \\ \mathit{match}(\Gamma', \emptyset, p_k, e_k) = \Gamma_k \quad \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_k \vdash^* e_k \Downarrow v_k \Rightarrow \Gamma'_k\end{array}}{\Delta;\Gamma \vdash^* \texttt{case } e \texttt{ do } p_1 \texttt{ -> } e_1\texttt{; } ...\texttt{; } p_k \texttt{ -> } e_n \texttt{ end} \Downarrow v_k \Rightarrow \Gamma'} \text{ (E\_CASE)}$$

Con la convención de que $[1,0] = \emptyset$, la regla anterior hace formal la descripción intuitiva de la Sección 2.5.6.3 para la semántica de la definición por casos. Luego de evaluar la expresión inicial, se escoge la primera rama que aplique exitosamente al patrón correspondiente, y se evalúa la expresión de la derecha enriqueciendo el contexto con las variables asignadas en dicho patrón. Finalmente, se olvidan las variables exportadas en la rama y únicamente se exporta el contexto de la primera evaluación.

El resto de las reglas para la evaluación exitosa de los operadores de control se pueden deducir por analogía a la definición por casos y se encuentran detallados en el Anexo D.2.

Para las llamadas a funciones declaradas, la regla que modela la evaluación correcta es:

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = (((p_1^1, \ldots, p_n^1), e^1), \ldots, ((p_1^m, \ldots, p_n^m), e^n)) \\ \Delta; \Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Delta; \Gamma \vdash^* e_n \Downarrow v_n \Rightarrow \Gamma_n \\ match(\emptyset, \emptyset, \{p_1^i, \ldots, p_n^i\}, \{v_1, \ldots, v_n\}) = \textbf{fail} \quad \forall i \in [1, \ldots, k-1] \\ match(\emptyset, \emptyset, \{p_1^k, \ldots, p_n^k\}, \{v_1, \ldots, v_n\}) = \Gamma^k \\ \Delta; \Gamma^k \vdash^* e^k \Downarrow v \Rightarrow \Gamma'\end{array}}{\Delta; \Gamma \vdash^* f\_name(e_1, \ldots, e_n) \Downarrow v \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (E\_DCALL)}$$

Aunque verbosa, el contenido de la regla anterior es sencillo: una vez evaluados los argumentos $e_1, \ldots, e_n$, se busca la primera instancia k de declaración cuyos parámetros $p_1^k, \ldots, p_n^k$ apliquen a los argumentos $v_1, \ldots, v_n$ y se evalúa el cuerpo de aquella instancia (e^k) con respecto al contexto Γ^k generado por el *pattern matching* secuencial de los parámetros respecto a los argumentos.

Finalmente, las dos reglas de evaluación correcta que involucran funciones anónimas son:

$$\frac{(f\_name, n) \in dom(\Delta)}{\Delta; \Gamma \vdash^* \&f\_name/n \Downarrow \&f\_name/n \Rightarrow \emptyset} \text{ (E\_ANON)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash^* e \Downarrow \&f\_name/n \Rightarrow \Gamma' \\ \Delta; \Gamma \vdash^* f\_name(e_1, \ldots, e_n) \Downarrow v \Rightarrow \Gamma''\end{array}}{\Delta; \Gamma \vdash^* e.(e_1, \ldots, e_n) \Downarrow v \Rightarrow \Gamma' \dagger \Gamma''} \text{ (E\_ACALL)}$$

4.2.2.3. Reglas con errores

Los errores en tiempo de ejecución contemplados para el fragmento son todos los de la sintaxis *Err* introducida previamente. Las reglas de evaluación que retornan errores se pueden clasificar en dos:

1. Generadoras de errores: Su conclusión es de la forma $\Delta; \Gamma \vdash^* e \Downarrow Err \Rightarrow \Gamma'$ cuando todas sus hipótesis de evaluación son de la forma $\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma'$.

2. Propagadoras de errores: Su conclusión es de la forma $\Delta; \Gamma \vdash^* e \Downarrow Err \Rightarrow \Gamma'$ cuando alguna hipótesis de evaluación es de la forma $\Delta; \Gamma \vdash^* e \Downarrow Err \Rightarrow \Gamma'$.

Las reglas generadoras de errores cubren las siguientes situaciones:

1. Si el argumento para la negación booleana (`not`) no es un booleano, se lanza un `ArgumentError`
2. Si el argumento para la negación numérica (`-`) no es un numérico, se lanza un `ArithmeticError`

3. Si alguno de los argumentos para +, *, -, / no es numérico, se lanza un `ArithmeticError`
4. Si alguno de los argumentos para `div`, `rem` no es entero, se lanza un `ArithmeticError`
5. Si alguno de los argumentos para <> no es un `string`, se lanza un `ArgumentError`
6. Si el segundo argumento para `and`, `or` no es booleano, se lanza un `BadBooleanError`
7. Si en un *pattern matching* el valor de la derecha no matchea con el patrón, se lanza un `MatchError`
8. Si en una definición por casos no aplica ninguna rama, se lanza un `CaseClauseError`
9. Si en una llamada no aplica ninguna instancia de declaración, se lanza un `FunctionClauseError`
10. Si en una llamada anónima el valor invocado no es una función, se lanza un `BadFunctionError`
11. Si en una llamada anónima el valor invocado es una función de aridad diferente al numero de argumentos, se lanza un `BadArityError`

A modo de ejemplo, la regla que corresponde al punto 6. es:

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \qquad match(\Gamma, \emptyset, p, v) = \textbf{fail}}{\Delta; \Gamma \vdash^* p \ = \ e \Downarrow \texttt{MatchError} \Rightarrow \emptyset} \text{ (E\_MATCH\_ERROR)}$$

Por otro lado, las reglas propagadoras únicamente se ocupan de promover estructuralmente el primer error encontrado evaluando las hipótesis. A modo de ejemplo:

$$\frac{\Delta; \Gamma \vdash^* e_i \Downarrow v_i \Rightarrow \Gamma_i \ \forall i \in \{1, .., k-1\} \qquad \Delta; \Gamma \vdash^* e_k \Downarrow Err \Rightarrow \Gamma_k}{\Delta; \Gamma \vdash^* \{e_1, \ldots, e_n\} \Downarrow Err \Rightarrow \emptyset}$$

Las reglas propagadoras de errores son muchas y poco interesantes. Por ejemplo, para el if/else se requieren tres, una cuando falla la condición, una cuando falla la rama if y otra cuando falla la rama else:

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow Err \Rightarrow \Gamma'}{\Delta; \Gamma \vdash^* \texttt{if}\ e\ \texttt{do}\ e_1\ \texttt{else}\ e_2\ \texttt{end} \Downarrow Err \Rightarrow \Gamma'}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \quad v \notin \{\texttt{:false}, \texttt{nil}\} \\ \Delta; \Gamma \dagger \Gamma' \vdash^* e_1 \Downarrow Err \Rightarrow \Gamma_1\end{array}}{\Delta; \Gamma \vdash^* \texttt{if}\ e\ \texttt{do}\ e_1\ \texttt{else}\ e_2\ \texttt{end} \Downarrow Err \Rightarrow \Gamma'}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \quad v \in \{\texttt{:false}, \texttt{nil}\} \\ \Delta; \Gamma \dagger \Gamma' \vdash^* e_2 \Downarrow Err \Rightarrow \Gamma_1\end{array}}{\Delta; \Gamma \vdash^* \texttt{if}\ e\ \texttt{do}\ e_1\ \texttt{else}\ e_2\ \texttt{end} \Downarrow Err \Rightarrow \Gamma'}$$

Debido a las dependencias entre los contextos de evaluación y los contextos exportados, aquí no resulta definible -al menos directamente- una semántica contextual como en Siek et al. [2015] que permite una presentación sucinta para todos estos casos. Se confía, de todos modos, en que el lector podrá deducir las reglas fácilmente si así lo quisiera.

4.2.3. Reglas para la evaluación de un programa

A diferencia de Cassola et al. [2022], la sintaxis del fragmento de estudio no incluye módulos ejecutables -es decir, módulos constituidos únicamente por una secuencia de expresiones-. Por eso, para la definición de la evaluación de un módulo `Program` aquí se asumirá que está definida una función `main/0` de una sola instancia. Dando por asumido esto último, la regla que define la evaluación de un programa m es:

$$\frac{\vdash^* m \Rightarrow \Delta \qquad \Delta[(\texttt{main}, 0)] = (((), e)) \qquad \Delta; \emptyset \vdash^* e \Downarrow r \Rightarrow \Gamma}{\vdash^* m \Downarrow r} \text{ (E\_PROG)}$$

siendo que el juicio de colección dinámica $\vdash^* m \Rightarrow \Delta'$ se define iterando secuencialmente la colección en todas sus declaraciones según describe el juicio auxiliar $\Delta \vdash^* d \Rightarrow \Delta'$:

$$\frac{\emptyset \vdash^* d_1 \Rightarrow \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash^* d_n \Rightarrow \Delta_n}{\vdash^* \texttt{defmodule Program do } d_1\texttt{, } \ldots\texttt{, } d_n \texttt{ end} \Rightarrow \Delta_n} \text{ (ECOLLECT\_PROG)}$$

y las siguientes reglas para $\Delta \vdash^* d \Rightarrow \Delta'$:

$$\frac{}{\Delta \vdash^* \texttt{@spec } f\_name(\tau_1, \ldots, \tau_n) \texttt{ :: } \tau_0 \Rightarrow \Delta} \text{ (EECOLLECT\_SPEC)}$$

$$\frac{(f\_name, n) \notin \Delta}{\Delta \vdash^* \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end} \Rightarrow \Delta[(f\_name, n) \mapsto (((p_1, \ldots, p_n), e))]} \text{ (ECOLLECT\_EDEF)}$$

$$\frac{\Delta[(f\_name, n)] = ((\vec{p}^1, e^1) \cdots, (\vec{p}^k, e^k))}{\Delta \vdash^* \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end} \Rightarrow \Delta[(f\_name, n) \mapsto ((\vec{p}^1, e^1) \cdots, (\vec{p}^k, e^k), ((p_1, \ldots, p_n), e))]} \text{ (ECOLLECT\_DEF)}$$

4.3. Evaluación en el lenguaje objeto

4.3.1. Introducción

Todas las reglas introducidas en la sección anterior para la definición de $\vdash^* m \Downarrow r$ siendo m un programa del lenguaje fuente se mantienen para el lenguaje objeto. Únicamente resta añadir las reglas que permiten reducir los casts.

Como se explicará en la sección siguiente, la representación del lenguaje objeto dentro de Elixir fue posible haciendo uso del mecanismo de macros nativo para simular los casts. Las limitaciones de este *approach* obligan a divergir de la mayoría de las definiciones de la literatura, y el motivo fundamental es que no resulta posible suspender la evaluación de los casts. Concretamente, la definición de los valores pasa a ser:

$$\begin{array}{rcl} v & ::= & l \mid \texttt{[ ]} \mid \texttt{[}v\texttt{|}v\texttt{]} \mid \texttt{\{}v\texttt{, } \ldots\texttt{, } v\texttt{\}} \mid \texttt{\%\{}l \texttt{ => } v\texttt{, } \ldots\texttt{, } l \texttt{ => } v\texttt{\}} \mid \texttt{\&}f\_name/arity \\ & \mid & (v \mid (T, .., T_n) \to T \rightsquigarrow (T, .., T_n) \to T) \end{array}$$

cuando lo estándar (por ejemplo, ver la Definición 2. de Cimini and Siek [2017]) es además incluir los casos:

$$
\begin{array}{rcl}
v & ::= & \cdots \\
 & | & (v \mid G \sim\!> T) \mid (v \mid [\,] \sim\!> [\,]) \mid (v \mid [T] \sim\!> [T]) \mid (v \mid \{T, \ldots, T_n\} \sim\!> \{T, \ldots, T_n\}) \\
 & | & (v \mid \%\{k_1 \Rightarrow T, \ldots, k_n \Rightarrow T\} \sim\!> \%\{k_1 \Rightarrow T, \ldots, k_n \Rightarrow T\})
\end{array}
$$

La sintaxis de los errores también se extiende para incluir el nuevo `CastError` cuando hay errores en la evaluación de *casts consistentes*, además del `BadCastError` para cuando los casts resultan inconsistentes:

$$
\begin{array}{rcl}
Err & ::= & TypeErr \mid \texttt{MatchError} \mid \texttt{CaseClauseError} \mid \texttt{FunctionClauseError} \\
 & | & \texttt{CastError} \mid \texttt{BadCastError}
\end{array}
$$

A continuación se introduce la noción de cast consistente y las funciones auxiliares $ground_V$ y $ground_T$ similares a $ground_P$ pero para valores y tipos respectivamente. Considerando esas dos definiciones, se procede a describir las nuevas reglas para el juicio de evaluación $\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma'$ en el lenguaje objeto.

4.3.2. Cast consistente

Recordar que el sistema de tipos con casts explícitos introduce una única regla T_CAST para el tratamiento de los casts. De acuerdo a esta regla, es posible afirmar que cada expresión chequeada de la forma $(e \mid \tau \sim\!> \sigma)$ necesariamente verifica $\tau \sim \sigma$. Por ejemplo, es posible ver que la siguiente *no* es una instancia de T_CAST:

$$
\frac{\Delta; \Gamma \vdash^t e : \texttt{integer} \Rightarrow \Gamma'}{\Delta; \Gamma \vdash^t (e \mid \texttt{integer} \sim\!> \texttt{float}) : \texttt{float} \Rightarrow \Gamma'}
$$

La intuición es que la instancia anterior debe ser rechazada porque ya se tiene la información para deducir que cualquier cast de la forma $(e \mid \texttt{integer} \sim\!> \texttt{float})$ fallará en tiempo de ejecución.

Otra restricción de la regla T_CAST está en que el tipo τ de la izquierda debe ser el inferido para e. Aunque durante la evaluación ese tipo pueda ser refinado a un subtipo[3], el subtipado estructural asegura que la "forma" del tipo se mantendrá tras la evaluación. La siguiente *tampoco* es una instancia de T_CAST:

$$
\frac{\Delta; \Gamma \vdash^t e : \{\tau, \tau\} \Rightarrow \Gamma'}{\Delta; \Gamma \vdash^t (e \mid \{\tau, \tau, \tau\} \sim\!> \texttt{any}) : \texttt{any} \Rightarrow \Gamma'}
$$

Las dos restricciones discutidas arriba se pueden combinar en la siguiente definición, que depende de las funciones auxiliares $ground_V : V \rightharpoonup G$ y $ground_T : \tilde{T} \to G$:

Cast consistente: Dados un valor v y dos tipos τ, σ, se dice que el cast $(v \mid \tau \sim\!> \sigma)$ es consistente si y solo si

1. $\tau \sim \sigma$
2. $ground_V(v) \lesssim ground_T(\tau)$

siendo que $ground_V$ y $ground_T$ están definidas por los procedimientos:

[3] por ejemplo, puede ser $\emptyset; [x \mapsto \texttt{number}] \vdash^t \texttt{x} : \texttt{number} \Rightarrow \emptyset$, mientras que $\emptyset; \Gamma \vdash^* \texttt{x} \Downarrow 1 \Rightarrow \emptyset$ siendo que $\Gamma = [x \mapsto 1]$ es adecuado y $\emptyset; [x \mapsto \texttt{number}] \vdash^t 1 : \texttt{integer} \Rightarrow \emptyset$

procedure GROUNDV(v)
  switch v **do**
    case l : **return** TYPE(l)
    case `[ ]` : **return** `[ ]`
    case `[`v`]` : **return** `[any]`
    case `{`v_1`,...,`v_n`}`
      return `{any,...,any}`
    case `%{`k_1 `=>` v_1`,...,`k_n `=>` v_n`}`
      return `%{`k_1 `=> any,...,`k_n `=> any}`
    case `&`*f_name/n*
      return $(\text{any}, .., \text{any}_n) \to \text{any}$
    case $(v \mid (\tau_1, .., \tau_n) \to \tau_0 \rightsquigarrow (\sigma_1, .., \sigma_n) \to \sigma_0)$
      return $(\text{any}, .., \text{any}_n) \to \text{any}$
    default : **fail**
end procedure

procedure GROUNDT(τ)
  switch τ **do**
    case b **when** $b \in B$: **return** b
    case `[ ]` : **return** `[ ]`
    case `[`τ`]` : **return** `[any]`
    case `{`τ_1`,...,`τ_n`}`
      return `{any,...,any}`
    case `%{`k_1 `=>` τ_1`,...,`k_n `=>` τ_n`}`
      return `%{`k_1 `=> any,...,`k_n `=> any}`
    case $(\tau_1, .., \tau_n) \to \tau_0$
      return $(\text{any}, .., \text{any}_n) \to \text{any}$
    case any
      return any
end procedure

Es posible establecer una correspondencia entre los tipos de G y los constructores de T; dado un elemento de la sintaxis se sustituyen todas las posiciones por instancias de `any`. Por ejemplo:

- `integer`^{0} (`integer` como constructor de aridad 0) se corresponde trivialmente con `integer` $\in G$
- $\{\cdot,\cdot\}^2$ (constructor de tuplas de aridad 2) se corresponde con `{any,any}` $\in G$
- $() \to \cdot^1$ (constructor de tipo funcional de 0 parámetros) se corresponde con el tipo $() \to$ `any` $\in G$

Bajo dicha correspondencia, $ground_V$ cumple el rol de extraer el constructor "más externo" del tipo de v:

- $\emptyset;\emptyset \vdash^t$ `1` : `integer` $\Rightarrow \emptyset$ y $ground_V(1) =$ `integer` $\approx$ `integer`^{0}
- $\emptyset;\emptyset \vdash^t$ `{1,2.0}` : `{integer,float}` $\Rightarrow \emptyset$ y $ground_V($`{1,2.0}`$) =$ `{any,any}` $\approx \{\cdot,\cdot\}^2$
- `(&`*f_name*`/0 |` $() \to$ `integer` $\rightsquigarrow () \to$ `float)` no se chequea correctamente y $ground_V($`(&`*f_name*`/n |` $() \to$ `integer` $\rightsquigarrow () \to$ `float)`$) =$ **fail**

Bajo la misma correspondencia, $ground_T$ puede verse como el extractor de la capa "más externa" del tipo τ, "olvidando" el resto de la estructura:

- $ground_T($`integer`$) =$ `integer` $\approx$ `integer`^{0}
- $ground_T($`{integer,float}`$) =$ `{any,any}` $\approx \{\cdot,\cdot\}^2$
- $ground_T(() \to$ `integer`$) = () \to$ `any` $\approx () \to \cdot^1$

Luego, la condición $ground_V(v) \lesssim ground_T(\tau)$ en el punto 2. de la definición se interpreta como una relación de compatiblidad entre los constructores externos de un valor v y su tipo τ. Por ejemplo:

1. $ground_V($`%{:one => 1,:two => 2.0}`$) =$ `%{:one => any,:two => any}`
2. $ground_V($`%{:one => float}`$) =$ `%{1 => any}`
3. `%{any => 1,any => float}` $\lesssim$ `%{1 => any}`

En general, la definición de cast consistente exige subtipado en vez de igualdad por el hecho de que el tipo asignado puede refinarse en *runtime* (como se mencionó arriba).

4.3.3. Nuevas reglas

La primera regla que se añade al sistema de evaluación es E_CAST, la cual permite propagar la evaluación de la expresión interna hacia toda la expresión casteada.

$$\frac{\Delta;\Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma'}{\Delta;\Gamma \dagger \Gamma' \vdash^* (v \mid \tau \leadsto \sigma) \Downarrow v' \Rightarrow \Gamma''}{\Delta;\Gamma \vdash^* (e \mid \tau \leadsto \sigma) \Downarrow v' \Rightarrow \Gamma' \dagger \Gamma''} \text{ (E\_CAST)}$$

Gracias a E_CAST, todas las reglas que siguen únicamente pueden asumir que la expresión interna al cast es un valor v. Aunque Γ'' debe introducirse como parte del juicio de evaluación del valor casteado, en ninguna de las nuevas reglas se exportan variables, por lo que a posteriori se puede afirmar que $\Gamma'' = \emptyset$ y $\Gamma' \dagger \Gamma'' = \Gamma'$.

La regla E_BAD_CAST_ERROR lanza un error ante cualquier ocurrencia de casts inconsistentes:

$$\frac{ground_V(v) \not\lesssim ground_T(\tau) \quad \text{ó} \quad \tau \nsim \sigma}{\Delta;\Gamma \vdash^* (v \mid \tau \leadsto \sigma) \Downarrow \texttt{BadCastError} \Rightarrow \emptyset} \text{ (E\_BAD\_CAST\_ERROR)}$$

Para expresiones correctamente chequeadas la condición $\tau \nsim \sigma$ es imposible por T_CAST, mientras que la condición $ground_V(v) \not\lesssim ground_T(\tau)$ es deseable y se conjetura como un invariante de la evaluación. De ser así -y siendo que ninguna otra regla genera errores de tipo `BadCastError`-, se espera poder garantizar que los errores de tipo `BadCastError` son imposibles dentro de código chequeado.

Para no solaparse con E_BAD_CAST_ERROR, el resto de las reglas deben exigir (tal vez implícitamente) que ambas condiciones $\tau \sim \sigma$ y $ground_V(v) \lesssim ground_T(\tau)$. Por ejemplo, en las reglas E_CAST_LIT y E_CAST_ELIST las hipótesis se incluyen únicamente para asegurar la consistencia del cast.

$$\frac{type(l) \leq \tau}{\Delta;\Gamma \vdash^* (l \mid \tau \leadsto \tau) \Downarrow l \Rightarrow \Gamma} \text{ (E\_CAST\_LIT)} \qquad \frac{[\,] \leq \tau_1 \quad [\,] \leq \tau_2 \quad \tau_1 \sim \tau_2}{\Delta;\Gamma \vdash^* ([\,] \mid \tau_1 \leadsto \tau_2) \Downarrow [\,] \Rightarrow \Gamma} \text{ (E\_CAST\_ELIST)}$$

Las reglas para los constructores de listas, tuplas y mapas asumen implícitamente la consistencia del cast al expresarse explícitamente en términos de los constructores externos de los valores y tipos involucrados. En los tres casos, los casts son "empujados" hacia dentro de las estructuras (puede ser de ayuda pensar en analogía con el *zipWith* de la programación funcional):

$$\frac{\Delta;\Gamma \vdash^* [(v_1 \mid \tau \leadsto \sigma) \mid (v_2 \mid [\tau] \leadsto [\sigma])] \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* ([v_1 \mid v_2] \mid [\tau] \leadsto [\sigma]) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_LIST)}$$

$$\frac{\Delta;\Gamma \vdash^* \{(v_1 \mid \tau_1 \leadsto \sigma_1), \ldots, (v_n \mid \tau_n \leadsto \sigma_n)\} \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (\{v_1, \ldots, v_n\} \mid \{\tau_1, \ldots, \tau_n\} \leadsto \{\sigma_1, \ldots, \sigma_n\}) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_TUPLE)}$$

$$\frac{J \subseteq I \qquad w_j = (v_j \mid \tau_j \leadsto \sigma_j) \quad \forall j \in J \qquad w_i = v_i \quad \forall i \in I \setminus J \qquad \Delta;\Gamma \vdash^* \%\{(\overline{k_i \Rightarrow w_i})_{i \in I}\} \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (\%\{(\overline{k_i \Rightarrow v_i})_{i \in I}\} \mid \%\{(\overline{k_i \Rightarrow \tau_i})_{i \in J}\} \leadsto \%\{(\overline{k_i \Rightarrow \sigma_i})_{i \in J}\}) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_MAP)}$$

Observar que la regla E_CAST_MAP es sutilmente más compleja que las otras dos ya que es posible para v tener más claves que τ y σ. De ser el caso, los valores correspondientes a las claves faltantes no interactúan con el cast y se preservan idénticas tras la evaluación. A continuación se expone un ejemplo:

$$\frac{\begin{array}{c} w_1 = \texttt{(\&f/1 | atom} \to \texttt{atom ~> any} \to \texttt{any)} \quad w_2 = \texttt{\&f/1} \\ \Delta;\emptyset \vdash^* \texttt{\%\{1 => } w_1\texttt{,2 => } w_2\texttt{\}} \Downarrow \texttt{\%\{1 => } w_1\texttt{,2 => } w_2\texttt{\}} \Rightarrow \emptyset \end{array}}{\begin{array}{c} \Delta;\emptyset \vdash^* \texttt{((\%\{1 => \&f/1,2 => \&f/1\} | \%\{1 => atom} \to \texttt{atom\}} \texttt{ ~> \%\{1 => any} \to \texttt{any\})} \\ \Downarrow \texttt{\%\{1 => (\&f/1 | atom} \to \texttt{atom ~> any} \to \texttt{any),2 => \&f/1\}} \Rightarrow \emptyset \end{array}}$$

Notar que la segunda hipótesis muestra como se evalúa `%{1 => `w_1`,2 => `w_2`}` sin cambios, y esto se debe a que tanto `&f/2` como `(&f/1 | atom → atom ~> any → any)` son valores del lenguaje (para un Δ adecuado).

Para explicar como se modifican las llamadas a funciones en presencia de casts, es necesaria la regla E_CAST_CALL :

$$\frac{\Delta;\Gamma \vdash^* (v.((v_1 \mid \sigma_1 \leadsto \tau_1), \ldots, (v_n \mid \sigma_n \leadsto \tau_n)) \mid \tau_0 \leadsto \sigma_0) \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (v \mid (\tau_1, .., \tau_n) \to \tau_0 \leadsto (\sigma_1, .., \sigma_n) \to \sigma_0).(v_1, \ldots, v_n) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_CALL)}$$

La regla anterior es estándar con respecto a la literatura (por ejemplo, Siek et al. [2015]) e indica como los casts pasan tanto al retorno como a los argumentos, invirtiendo los roles de los tipos involucrados en el segundo caso. Para motivar, se propone un ejemplo con la función `identity` de tipo `number → any`:

```
@spec identity(number) :: any
def identity(x) do
    x
end
```

En presencia del cast hacia `any → integer`, el comportamiento es el siguiente:

- `(&identity/1 | number → any ~> any → integer).(:one)` retorna `CastError` porque el cast del argumento `(:one | any ~> number)` falla.
- `(&identity/1 | number → any ~> any → integer).(1.0)` retorna `CastError`:
  1. El cast del argumento `(1.0 | any ~> number)` es exitoso
  2. `&identity/1.(1.0)` evalúa a `1.0`
  3. El cast del retorno `(1.0 | any ~> integer)` falla.
- `(&identity/1 | number → any ~> any → integer).(1)` evalúa a `1`:
  1. El cast del argumento `(1 | any ~> number)` es exitoso
  2. `&identity/1.(1)` evalúa a `1`
  3. El cast del retorno `(1 | any ~> integer)` es exitoso.

Las cuatro reglas que siguen tratan los casos donde `any` ocurre como alguno de los tipos involucrados en el cast.

$$\frac{\Delta;\Gamma \vdash^* v \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (v \mid \texttt{any} \leadsto \texttt{any}) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_ANY\_ANY)}$$

$$\frac{\tau \neq \texttt{any} \qquad \Delta;\Gamma \vdash^* (v \mid \tau \leadsto \mathit{ground}_T(\tau)) \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (v \mid \tau \leadsto \texttt{any}) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_GROUND\_ANY)}$$

$$\frac{\tau \neq \texttt{any} \quad \mathit{ground}_V(v) \lesssim \mathit{ground}_T(\tau) \qquad \Delta; \Gamma' \vdash^* (v \mid \mathit{ground}_V(\tau) \sim\!> \tau) \Downarrow w \Rightarrow \Gamma'}{\Delta; \Gamma \vdash^* (v \mid \texttt{any} \sim\!> \tau) \Downarrow w \Rightarrow \Gamma'} \text{ (E\_CAST\_ANY\_GROUND)}$$

$$\frac{\sigma \neq \texttt{any} \quad \mathit{ground}_V(v) \not\lesssim \mathit{ground}_T(\sigma)}{\Delta; \Gamma \vdash^* (v \mid \texttt{any} \sim\!> \sigma) \Downarrow \texttt{CastError} \Rightarrow \emptyset} \text{ (E\_CAST\_CAST\_ERROR)}$$

Las reglas anteriores siguen siendo dirigidas por sintaxis ya que los tipos del cast forman parte de la expresión evaluada. Para brindar una mayor intuición, se presenta a continuación una explicación en lenguaje natural:

1. Los casts de la forma $(v \mid \texttt{any} \sim\!> \texttt{any})$ son trivialmente exitosos.
2. Los casts de la forma $(v \mid \tau \sim\!> \texttt{any})$ se refinan a $(v \mid \tau \sim\!> \mathit{ground}_T(\tau))$, lo cual no compromete la consistencia ya que $\tau \sim \mathit{ground}_T(\tau)$.
3. Los casts de la forma $(v \mid \texttt{any} \sim\!> \sigma)$ se intentan reemplazar por $(v \mid \mathit{ground}_T(\sigma) \sim\!> \sigma)$. Si esto fuera a comprometer la consistencia del cast ($\mathit{ground}_V(v) \not\lesssim \mathit{ground}_T(\sigma)$), se lanza un `CastError`. De lo contrario el reemplazo se hace efectivo.

Para ejemplificar el uso de las reglas anteriores, se justificará en detalle la validez del siguiente juicio de evaluación:

$$\Delta; \emptyset \vdash^* ((\texttt{\&foo/2} \mid (\tau, \tau) \to \tau \sim\!> \texttt{any}) \mid \texttt{any} \sim\!> \texttt{any} \to \texttt{any}) \Downarrow \texttt{CastError} \Rightarrow \emptyset$$

siendo que

$$\Delta = [(\texttt{foo}, 2) \mapsto ((\tau, \tau), \tau)]$$

Intuitivamente, el `CastError` para esta expresión viene del hecho de que el cast interno $(\cdot \mid (\tau, \tau) \to \tau \sim\!> \texttt{any})$ oculta la aridad real (2) en el tipo de `&foo/2`; mientras que el cast externo $(\cdot \mid \texttt{any} \sim\!> \texttt{any} \to \texttt{any})$ intenta forzar el tipo de la expresión hacia una función de aridad 1. Durante la evaluación, la información de tipos se refinará hasta descubrir la inconsistencia entre la aridad real y la forzada. A continuación se describe la evaluación formal para este caso:

1. La primera regla (en el sentido *bottom-up*) aplicada es E_CAST, que reduce $(\texttt{\&foo/2} \mid (\tau, \tau) \to \tau \sim\!> \texttt{any})$ hacia un valor, y luego lo inserta en el cast externo que termina lanzando un `CastError`:

$$\frac{\Delta; \emptyset \vdash^* (\texttt{\&foo/2} \mid (\tau, \tau) \to \tau \sim\!> \texttt{any}) \Downarrow (\texttt{\&foo/2} \mid (\tau, \tau) \to \tau \sim\!> (\texttt{any}, \texttt{any}) \to \texttt{any}) \Rightarrow \emptyset \qquad \Delta; \emptyset \vdash^* ((\texttt{\&foo/2} \mid (\tau, \tau) \to \tau \sim\!> (\texttt{any}, \texttt{any}) \to \texttt{any}) \mid \texttt{any} \sim\!> \texttt{any} \to \texttt{any}) \Downarrow \texttt{CastError} \Rightarrow \emptyset}{\Delta; \emptyset \vdash^* (\texttt{x} \mid \texttt{any} \sim\!> \texttt{any} \to \texttt{any}) \Downarrow \texttt{CastError} \Rightarrow \emptyset}$$

2. La primera hipótesis de la regla anterior se justifica con E_CAST_GROUND_ANY, que refina la información del tipo de la derecha en el cast interno:

$$\frac{\mathit{ground}_T((\tau, \tau) \to \tau) = (\texttt{any}, \texttt{any}) \to \texttt{any} \qquad \Delta; \emptyset \vdash^* (\texttt{\&foo/2} \mid (\tau, \tau) \to \tau \sim\!> (\texttt{any}, \texttt{any}) \to \texttt{any}) \Downarrow (\texttt{\&foo/2} \mid (\tau, \tau) \to \tau \sim\!> (\texttt{any}, \texttt{any}) \to \texttt{any}) \Rightarrow \emptyset}{\Delta; \emptyset \vdash^* (\texttt{\&foo/2} \mid (\tau, \tau) \to \tau \sim\!> \texttt{any}) \Downarrow (\texttt{\&foo/2} \mid (\tau, \tau) \to \tau \sim\!> (\texttt{any}, \texttt{any}) \to \texttt{any}) \Rightarrow \emptyset}$$

3. El `CastError` se descubre al intentar evaluar el cast externo con la información refinada del paso anterior y descubrir la incompatibilidad entre tipos funcionales de distinta aridad:

$$\frac{\begin{array}{c}(\text{any},\text{any}) \to \text{any} \neq \text{any} \\ \mathit{ground}_V(\texttt{(\&foo/2} \mid (\tau,\tau) \to \tau \rightsquigarrow (\text{any},\text{any}) \to \text{any})) = (\text{any},\text{any}) \to \text{any} \\ \mathit{ground}_T(\text{any} \to \text{any}) = \text{any} \to \text{any} \\ (\text{any},\text{any}) \to \text{any} \not\lesssim \text{any} \to \text{any}\end{array}}{\Delta;\emptyset \vdash^* \texttt{((\&foo/2} \mid (\tau,\tau) \to \tau \rightsquigarrow (\text{any},\text{any}) \to \text{any}) \mid \text{any} \rightsquigarrow \text{any} \to \text{any}) \Downarrow \texttt{CastError} \Rightarrow \emptyset}$$

4.4. Evaluación gradual para programas chequeados

En esta sección se define la evaluación gradual para programas correctos respecto al sistema de tipos del capítulo anterior. A diferencia de la evaluación como fue descrita informalmente en 2 y luego establecida rigurosamente en 4.2, la nueva semántica de evaluación se restringe únicamente a programas válidos y aprovecha la información de los tipos para prevenir las inconsistencias en tiempo de ejecución dentro del código anotado estáticamente.

Siguiendo a Siek et al. [2015], la evaluación gradual de un módulo chequeado se define indirectamente como la secuencia entre una traducción al lenguaje con casts según la definición de la Sección 4.1 y la evaluación del módulo resultado de dicha traducción utilizando las reglas de la Sección 4.3 para los casts:

$$\vdash^* m \,\tilde{\Downarrow}\, v ::= m \rightsquigarrow m' \;\wedge\; \vdash^* m' \Downarrow v \qquad \text{(evaluación gradual)}$$

Dicho en otras palabras, la evaluación gradual $\vdash^* m \,\tilde{\Downarrow}\, v$ para un programa válido m procede insertando casts según $\vdash m \rightsquigarrow m'$ y luego evaluando el resultado m' según $\vdash^* m' \Downarrow v$, contemplando las nuevas reglas para los casts. Notar que para que la definición tenga sentido es necesario garantizar que m' está definido, y esto está garantizado para cualquier m válido gracias a la correctitud estática de la traducción enunciada en la Sección 4.1.6.

Para todos los ejemplos a continuación, se considera un único módulo `Program` para el cual se instancia en cada caso el argumento de `map_succ` dentro de la declaración de `main/0`:

```
defmodule Program do
  @spec map((any -> any), [any]) :: [any]
  def map(f, l) do
    case l do
      [] -> []
      [head | tail] -> [f.(head) | map(f, tail)]
    end
  end

  @spec succ(number) :: number
  def succ(x) do
    x + 1
  end

  @spec map_succ([number]) :: [number]
  def map_succ(l) do
    map(&succ/1, l)
  end

  def untyped(x) do
```

```
    x
  end

  def main do
    map_succ(...)
  end
end
```

Como la traducción $\vdash m \rightsquigarrow m'$ es estructural, ya es posible obtener el resultado de la traducción a menos del argumento de `map_succ` dentro de `main/0`. Los únicos cambios ocurren dentro del argumento de `map_succ`:

```
defmodule Program do
  use UseCast

  @spec map((any -> any), [any]) :: [any]
  def map(f, l) do
    case l do
      [] -> []
      [head | tail] -> [f.(head) | map(f, tail)]
    end
  end

  @spec succ(number) :: number
  def succ(x) do
    x + 1
  end

  @spec map_succ([number]) :: [number]
  def map_succ(l) do
    map((&succ/1) | (number -> number) ~> (any -> any), l | [number] ~> [any]) | [any] ~> [number]
  end

  def untyped(x) do
    x
  end

  def main() do
    map_succ(...) | [number] ~> any
  end
end
```

Ejemplo 1:

```
map_succ([1 | [-2 | [3 | []]]])
```

1. La evaluación estándar concluye exitosamente retornando `[2|[-1|[4|[]]]]`.
2. La expresión se traduce sin cambios ya que `[1|[-2|[3|[]]]]` es de tipo `[integer]` y `[integer]` $\leq$ `[number]`, siendo que `[number]` es el tipo para el argumento de `map_succ`.
3. De los dos puntos anteriores, resulta claro que la evaluación gradual coincide con la estándar.

Ejemplo 2:

```
map_succ([untyped(1) | [[]]])
```

1. La evaluación estándar concluye exitosamente retornando `[2|[]]`.
2. La expresión se traduce a:

   ```
   map_succ([untyped(1 | integer ~> any) | []] | [any] ~> [number])
   ```

3. Como los casts resultan exitosos (también los internos a `map_succ`), el resultado coincide con la evaluación estándar.

Ejemplo 3:

```
map_succ([untyped(true) | [[]]])
```

1. La evaluación estándar concluye lanzando un `ArithmeticError` por intentar utilizar `true` como sumando izquierdo dentro de `succ`.
2. La expresión se traduce a:

   ```
   map_succ([untyped(true | true ~> any) | []] | [any] ~> [number]) | [number] ~> any
   ```

3. La evaluación gradual lanza un `CastError` al evaluar el cast `([true|[]] | [any] ~> [number])`, por lo que ni siquiera se llega a realizar la suma.

Ejemplo 4:

```
map_succ(untyped(true))
```

1. La evaluación estándar concluye lanzando un `CaseClauseError` por no "matchear" ninguna rama para `true` dentro del `case` en la declaración de `map`.
2. La expresión se traduce a:

```
    map_succ(untyped(true | true ~> any) | any ~> [number]) | [number] ~> any
```

3. La evaluación gradual lanza un `CastError` al evaluar el cast `(untyped(true) | any ~> [number])`, por lo que ni siquiera se llega a realizar la suma.

Capítulo 5

Implementación

La implementación del sistema de tipos y la evaluación gradual se puede encontrar en el repositorio de github del proyecto[1]. Allí se encuentra tanto el código fuente como los pasos para descargar e instalar localmente la herramienta de línea de comandos `gradualelixir`. Ésta ofrece cuatro modos de interacción:

- `type_check`: Ejecuta el chequeo para un archivo de Elixir que contenga un único módulo. El chequeo por defecto es bajo la modalidad estática; la versión gradual se activa agregando la *flag* `--gradual`.
- `type_check --annotate casts`: Genera un nuevo archivo que contiene el código objeto de la traducción para un archivo de Elixir correspondiente a un módulo correcto.
- `run`: Ejecuta la función `main/0` definida en un archivo de Elixir que contenga un único módulo que la incluya. La semántica de evaluación gradual descrita en la Sección 4.4 se activa agregando la flag `--gradual`.
- `test --display-results`: Ejecuta la suite de tests unitarios para cualquiera de los componentes involucrados en el chequeo y la traducción, exhibiendo el contenido de cada test en un formato amigable. El contenido de la suite se describe en la Sección 5.3.

Para la implementación de la rutina de chequeo y traducción se utilizó el lenguaje de programación Python y se utilizó Elixir para la definición de la macro para casts y la ejecución dinámica de `main/0`.

5.1. Descripción de la arquitectura

La implementación de la rutina de chequeo se realiza dentro de la función `type_check` del módulo Python `module.py`. Esta función toma como entrada la representación interna de un módulo Elixir y ejecuta, algorítmicamente, las reglas de chequeo para cada declaración de función tomando en cuenta las especificaciones declaradas. A su vez, para el chequeo se invocan las subrutinas homónimas `type_check` para el chequeo de expresiones y patrones ubicadas dentro de los módulos `expression.py` y `pattern.py` respectivamente.

La traducción al lenguaje con casts se implementa como un *consumidor* de la rutina de chequeo, recibiendo de ella un argumento `type_derivation`: `TypeCheckSuccess` que encapsula, en una estructura de datos arborescente, toda la información del chequeo exitoso.

El módulo `elixir_port.py` es el responsable de la comunicación con los módulos Elixir:

[1] bajo el url https://github.com/damif94/gradualelixir

- `AstTransformer`: recibe el contenido de un módulo Elixir (en un string) y devuelve la representación AST correspondiente en formato JSON. `elixir_port.py` parsea y transforma dicho AST a su representación interna dentro de Python.
- `Runner`: recibe el contenido de un módulo Elixir (en un string) y luego de compilar dinámicamente, ejecuta la función `main/0`. En este caso, `elixir_port.py` únicamente sirve como un pasamanos, devolviendo en un string el *output* recibido de Elixir.

Un detalle importante es que todos los módulos Elixir del proyecto están consolidados dentro de un paquete *Mix*[2], por lo que la compilación dinámica dentro de `Runner` podrá referenciar al módulo interno `UseCast` que contiene la definición de la primitiva de cast (explicado en la sección siguiente).

En la Figura 5.1 se presenta un diagrama de flujo que ilustra el proceso de ejecución de `run program.ex`, para la evaluación estándar (no gradual) de la función `main/0` dentro del módulo definido en el archivo `program.ex`. Observar que en este caso no se incluyen las rutinas de chequeo ni traducción, y únicamente se invoca al `elixir_port` para validar que el módulo pertenece al fragmento de estudio y que contiene al menos una declaración para `main/0`.

Para la evaluación gradual, el proceso se ilustra en la Figura 5.2. Éste supone mayor complejidad que el anterior, ya que se incluyen nuevas etapas para el chequeo y posterior traducción en caso de no haber errores en el primero. Además, el proceso `Runner` recibirá un módulo que posiblemente incluya casts; es por esto que se añade al comienzo la directiva `use UseCast` que inyecta la definición de la macro correspondiente.

Las rutinas de Python `module.type_check`, `expression.type_check`, `pattern.type_check` implementan los algoritmos de decisión para los juicios j. chequeo de programa, j. chequeo de expresión y j. chequeo-refinamiento de patrones respectivamente. Esto se visualiza fácilmente comparando una a una las signaturas de los juicios con las firmas de las rutinas:

Para decidir $\vdash^t m$, se tiene la rutina `module.type_check`:

```
def type_check(
    module: Module, static: bool
) -> t.Union[CollectSpecsResultErrors, TypeCheckErrors, TypeCheckSuccess]:
    ...
```

que decide si el módulo `module` es correcto o falla en una de las dos formas:

1. `CollectSpecsResultErrors`: Cuando la falla ocurre decidiendo por $\vdash^c \Delta \Rightarrow m$. Esto es fuertemente dependiente de la flag `gradual` que indica la modalidad del chequeo (estática o gradual).

2. `TypeCheckErrors`: Cuando la falla viene del chequeo $\Delta \vdash^t d$ en alguna de las especificaciones, propagándose hacia una falla en $\Delta \vdash^{ts} m$.

En caso de ser correcto, `TypeCheckSuccess` contiene toda la información de la derivación construida para el juicio.

Para decidir $\Delta; \Gamma \vdash^t e : \tau \Rightarrow \Gamma'$, se tiene la rutina `expression.type_check`:

[2]herramienta de empaquetado oficial de Elixir

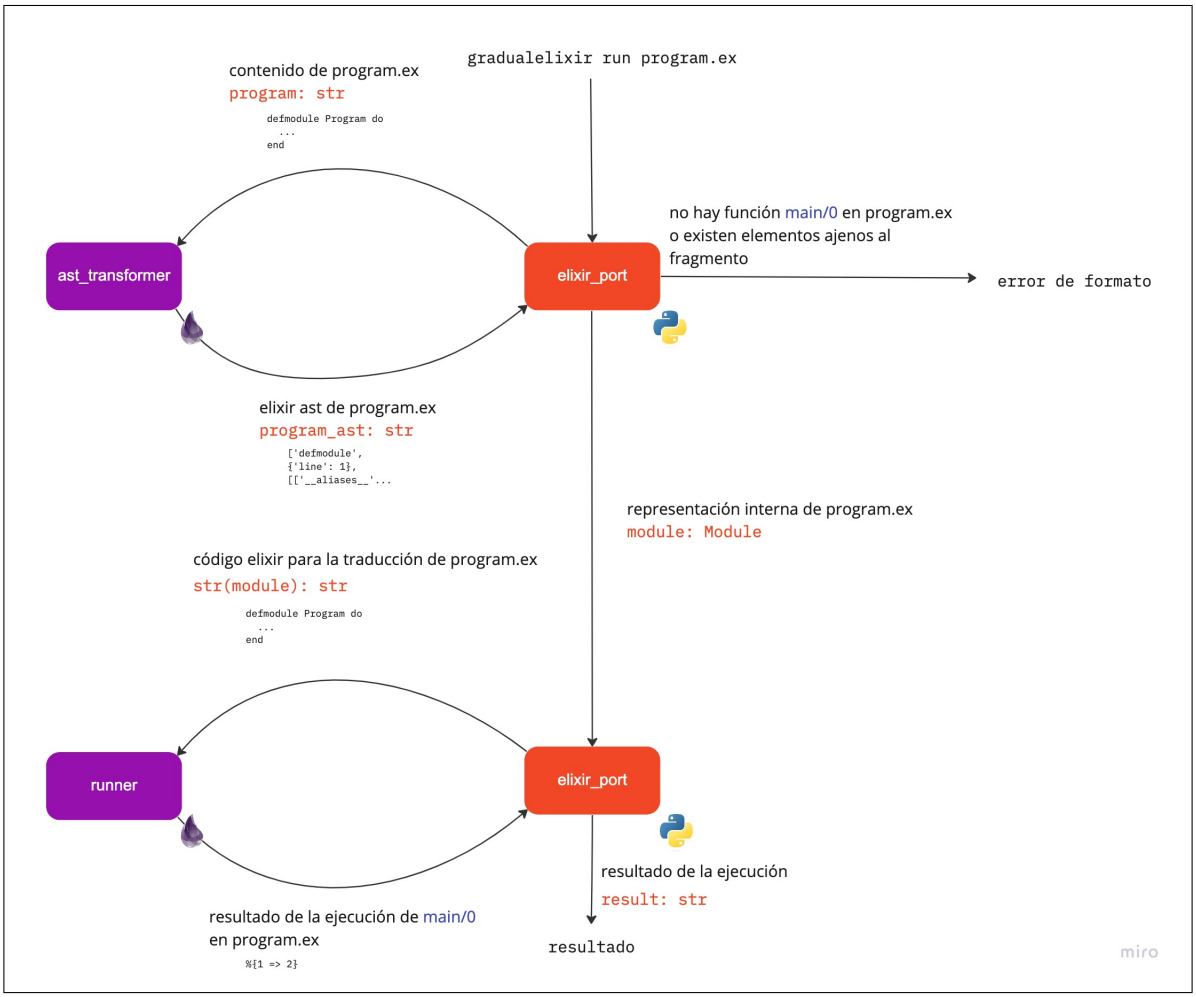


Figura 5.1: Diagrama de la arquitectura de la evaluación estándar

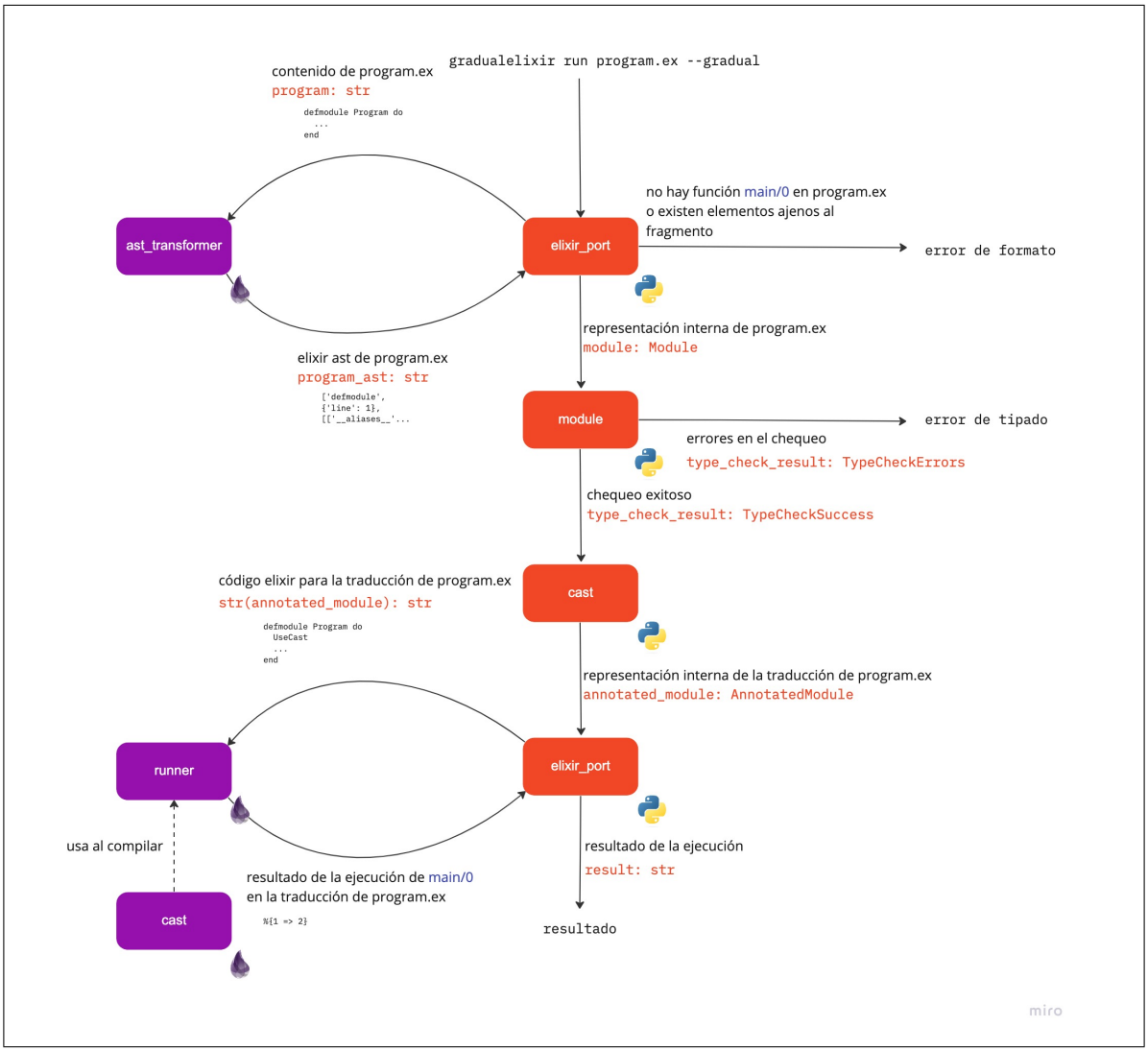


Figura 5.2: Diagrama de la arquitectura de la evaluación gradual

```
    def type_check(
        expr: Expression, env: gtypes.TypeEnv, specs_env: gtypes.SpecsEnv
    ) -> t.Union[ExpressionTypeCheckError, ExpressionTypeCheckSuccess]:
        ...
```

que, de manera similar, devuelve en `ExpressionTypeCheckSuccess` una estructura con la información de la derivación del juicio de arriba, o una falla en `ExpressionTypeCheckError`.

Finalmente, para decidir $\Sigma; \tau \vdash^{pt} p : \sigma \Rightarrow \Gamma'$, se tiene la rutina `pattern.type_check`:

```
def type_check(
    pattern: Pattern, type: gtypes.Type, env: gtypes.TypeEnv, external_env: gtypes.TypeEnv,
) -> t.Union[PatternMatchSuccess, PatternMatchError]:
    collect_env_result = collect_env(pattern, type, env, external_env)
    if isinstance(collect_env_result, PatternMatchError):
        return collect_env_result
    refine_types_result = refine_type(pattern, type, collect_env_result, external_env)
    if isinstance(refine_types_result, PatternMatchError):
        return refine_types_result
    return PatternMatchSuccess(pattern, type, external_env, env, refine_types_result, collect_env_result)
```

En la definición de arriba, el parámetro `env` puede introducirse no vacío para poder hacer testeos unitarios. Por el resto, notar que la definición refleja -módulo las particularidades sintácticas de Python y el orden de los parámetros- la estructura de la regla TCP para el j. chequeo-refinamiento de patrones:

$$\frac{\Gamma; \emptyset; \tau \vdash p \Rightarrow \Gamma'; \Sigma' \qquad \Sigma'; \Gamma'; \tau \vdash p : \sigma}{\Gamma; \tau \vdash^{pt} p : \sigma \Rightarrow \Gamma} \text{(TCP)}$$

La rutina `cast.translate_module` implementa el j. traducción de programa. Mientras que la signatura del juicio en cuestión es $\vdash m \Rightarrow m'$, la firma de la rutina Python es:

```
def translate_module(type_derivation: module.TypeCheckSuccess, casts: bool) -> AnnotatedModule:
    ...
```

siendo `type_derivation` una representación interna -como ya se explicó- de la derivación D encontrada para el juicio $\vdash m$. En términos de juicios, es posible resumir esquemáticamente su funcionamiento como:

$$\underset{\vdash m}{D} \xRightarrow{\texttt{translate\_module}} m' \;/\; \exists D', \underset{\vdash m \rightsquigarrow m'}{D'}$$

Aunque resulta inadecuado para una implementación escalable (D crece en tamaño aproximadamente lineal respecto al número de líneas de m), utilizar D en vez de m resulta ventajoso en este punto de la implementación porque permite mantener las rutinas de chequeo aisladas del proceso de traducción y viceversa.

5.2. Descripción de la macro para casts

En esta sección se describe la implementación de la macro `(· | · ~> ·)` que permitió representar los casts de tipo con la semántica de la Sección 4.3 de manera nativa. En particular, esto fue llevado a cabo utilizando algunas de las herramientas que Elixir provee para la metaprogramación:

1. Las macros de Elixir (se puede leer la introducción oficial en *Macros*) permiten escribir funciones que transforman el AST previo a la compilación.

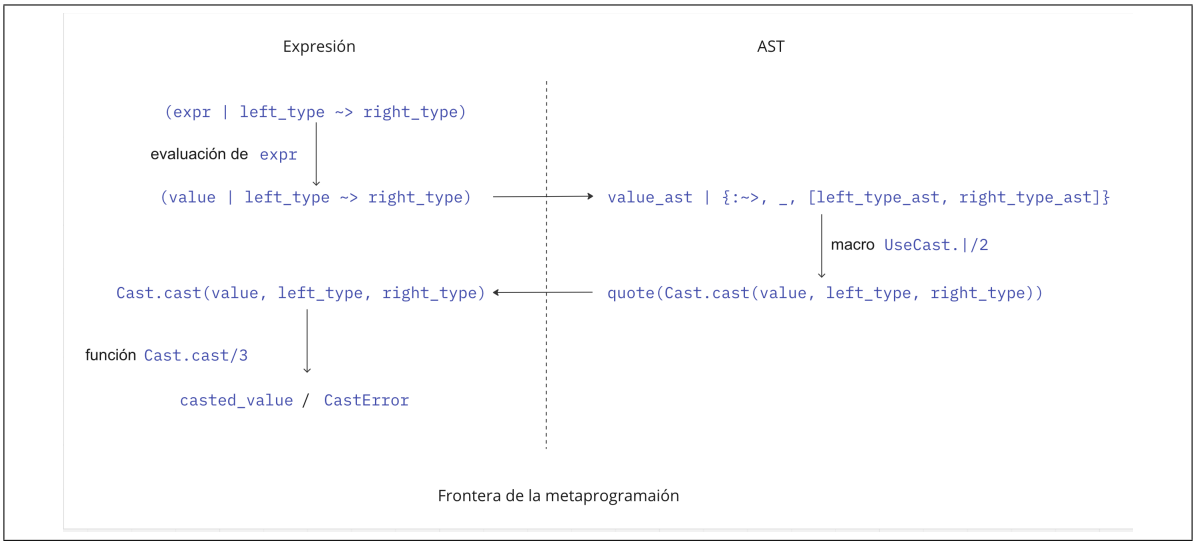
2. La macro reservada `use` que permite que ciertos módulos especiales (aquellos que definan la macro `__using__`) sean inyectados en el AST de otro módulo (ver la documentación oficial en *Alias, Require and Imports*).

3. La directiva `bind_quoted` que permite posponer la evaluación de una macro a tiempo de ejecución y utilizar el AST de las expresiones declaradas dentro del `quote` únicamente *luego* de que éstas son evaluadas. Una buena referencia (no oficial) para leer sobre esta directiva es el *blogpost* *Understanding Elixir Macros, part 3*.

La macro de cast está definida en el módulo Elixir `UseCast` (dentro del archivo `elixir_port/lib/cast.ex`) utilizando el operador binario "`|`", tomando el AST de un valor `value` para el primer parámetro y asumiendo en el segundo la forma del AST del cast desde `left_type` hacia `right_type`:

```
defmacro value_ast | {:~>, _, [left_type_ast, right_type_ast]} do
    ... # se liga value_ast dentro de context para esperar a ser evaluado
    quote bind_quoted: [context: context] do
      ... # se extrae value_ast de context
      Cast.cast(value_ast, left_type, right_type)
    end
end
```

Gracias a la variante `bind_quoted` para la metaprogramación, la ejecución de cualquier expresión casteada `(expr | left_type ~> right_type)` se pospone al momento en que `expr` es evaluado -por ejemplo, hacia un valor `value`-. En ese momento, la expresión `(value | left_type ~> right_type)` se entrega a la macro `UseCast.|/2` que la transforma en la expresión `Cast.cast(value_ast, left_type, right_type)`. Al evaluarse, esta última podrá devolver un valor casteado `casted_value` o un error `CastError`. La explicación anterior se esquematiza en la siguiente figura:



El cast en sí mismo está definido en la función `cast/3` del módulo `Cast`. Esta función se implementa siguiendo las reglas introducidas para la evaluación de los casts en la Sección 4.3. El caso más interesante es el de los casts entre tipos funcionales: se recuerda que las expresiones de la forma $(v \mid (\tau_1, .., \tau_n) \to \tau_0 \sim> (\sigma_1, .., \sigma_n) \to \sigma_0)$ deben compilar a un valor del lenguaje con la semántica definida por E_CAST_CALL. Esto se consigue envolviendo la función v pasada como argumento dentro de una función anónima de Elixir[3] que castea los argumentos y el valor retornado. Por ejemplo:

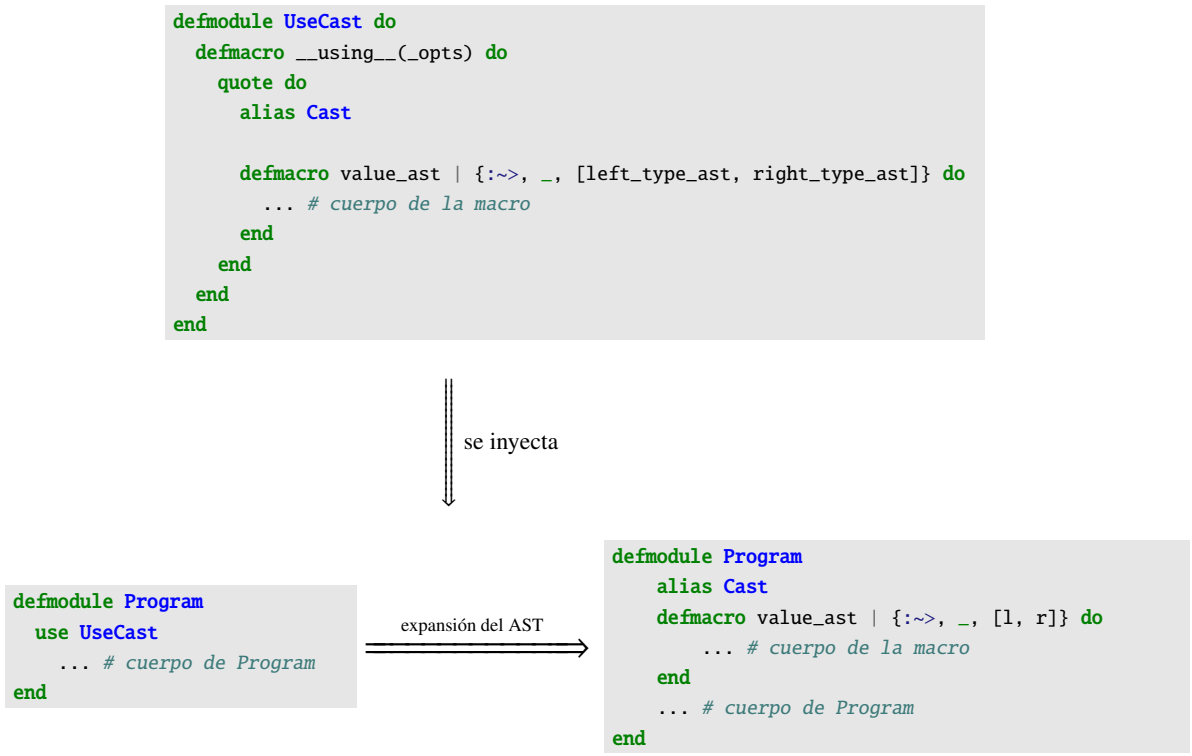
```
Cast.cast(&foo/1, number ~> any, any ~> integer)
```

evalúa a

```
fn arg ->
    casted_arg = Cast.cast(arg, any, number)
    ret = &foo/1.(casted_arg)
    casted_ret = Cast.cast(arg, any, integer)
    casted_ret
end
```

El archivo `cast.ex` contiene la definición de los módulos `Cast` y `UseCast`, siendo el primero una dependencia del segundo. Además, `UseCast` define la macro `|/2` dentro del cuerpo de `__using__`; esto significa que cualquier otro módulo que comience con la sentencia `use UseCast` será inyectado con la definición de la macro `|/2` en la etapa de expansión del AST.

[3] fuera del fragmento definido para el trabajo



5.3. Descripción de la suite de tests

La implementación del proyecto está acompañada de una suite de tests que cumple dos funciones:

1. Brindar calidad a la implementación final y garantizar robustez en los momentos en que se requieren cambios.
2. Servir como documentación de la implementación.

A efectos de potenciar el uso de los tests como documentación, la suite se integra a la herramienta de línea de comandos para poder ofrecer una visualización completa y amigable de todos los escenarios incluidos. Una vez instalado el proyecto, cada uno de los grupos de la suite se puede visualizar en la consola ejecutando `gradualelixir test --include [grupo]`, siendo los grupos:

- `gtypes`: contiene escenarios correctos e incorrectos para las funciones a nivel de tipos (12 subgrupos).
- `pattern`: contiene escenarios correctos e incorrectos para el j. chequeo-refinamiento de patrones (129 en total).
- `expression`: contiene escenarios correctos e incorrectos para el j. chequeo de expresión (168 en total).
- `cast`: contiene escenarios correctos para el j. traducción de programa (40 en total).

Además de los anteriores, hay un grupo de tests para la implementación del cast (12 subgrupos) -aunque éstos no forman parte de la suite-. Éstos se encuentran dentro del archivo `elixir_port/test/cast.ex`.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

En este trabajo se introdujo un sistema de tipos gradual para un fragmento de Elixir que brinda garantías en *runtime* respecto a los chequeos estáticos, llevando un poco más lejos los esfuerzos tendientes al desarrollo de un sistema de tipos que se adecue a las necesidades de la comunidad del lenguaje.

Para la construcción del sistema del sistema de tipos, se ha buscado un equilibrio entre formalidad y expresividad. Se han tenido en cuenta los aportes acumulados por varios años de investigación en el área, en particular en lo que refiere al diseño de sistemas graduales con subtipado estructural, con el fin de fundamentar la correctitud de la implementación. La expresividad, tanto del sistema como del fragmento considerado, ha debido limitarse en gran parte por las dificultades que supone mantener ese rigor en el contexto de un proyecto de grado.

Por lo pronto, en éste trabajo se ha demostrado que es posible implementar un sistema gradual que introduce chequeos por consistencia en el *runtime* de Elixir sin ser intrusivos en la etapa de compilación y basándose únicamente en el contenido del código fuente, aprovechando para el mecanismo de macros del lenguaje para extender la sintaxis.

Lo que se presenta en el trabajo actual está aún lejos de poder usarse en un proyecto real de Elixir. Por un lado, el fragmento del lenguaje es considerablemente limitado respecto al lenguaje completo y adaptar el sistema a todo el lenguaje supone extenderlo en dos direcciones: más reglas para nuevas construcciones sintácticas y un sistema de tipos más rico que permita integrar correctamente a algunas de esas construcciones. Por otro lado, el tamaño de los proyectos en un contexto real obliga a tomar consideraciones de eficiencia, tanto en el sistema estático como en la semántica operacional de los casts. Respecto a esto último, un paso necesario es redefinir la evaluación de los casts de manera de evitar un crecimiento innecesario del *stack* de llamadas (ver Siek and Wadler [2010]).

6.2. Trabajo Futuro

A continuación se exponen algunas ideas que han surgido durante la realización del trabajo, pero por cuestiones de alcance han sido postergadas para su consideración. Los aportes que podrían surgir de ellas van en dos direcciones:

1. La profundización teórica en la formalización actual, con el fin de garantizar la verificación de nuevas propiedades formales (como la *Dynamic Gradual Guarantee* y *type safety*).

2. Refinar el sistema de tipos, en el sentido de que se obtenga un mayor número de programas chequeados sin comprometer la correctitud formal.

6.2.1. Operadores básicos polimórficos

A diferencia de Cassola et al. [2022], en este trabajo se omitieron los operadores polimórficos ++ y $e[k]$ que definen la concatenación de listas y el acceso a mapas por clave k respectivamente. Para poder integrarlos correctamente al sistema de este trabajo, una idea que parece funcionar es la de adelantar la presentación de la sintaxis de los tipos ground G y el subtipado-precisión ($\prec^-$) y generalizar la propiedad 3. enunciada para los operadores básicos (del Argumento máximo) por:

> Para cada $\oplus$, existen $\tau_1^\oplus, \tau_2^\oplus \in G$ tales que $binOpType(\oplus, \tau_1^\oplus, \tau_2^\oplus)$ está definido y para cualquier otro par $\tau_1, \tau_2 \in T$ tal que $binOpType(\oplus, \tau_1, \tau_2)$ está definido, $\tau_1 \prec^- \tau_1^\oplus$ y $\tau_2 \prec^- \tau_2^\oplus$

Luego, observar que definiendo

procedure ConcatType(τ, σ)
  switch τ, σ **do**
    case `[`τ`]`, `[`σ`]`
      return `[`Sup(τ, σ)`]`
    default
      fail
end procedure

procedure MapAccessType(k, τ)
  switch τ **do**
    case `%{`$\overline{(k_i \texttt{ => } \tau_i)_{i \in I}}$`}`
      for k_0, τ_0 **in** `%{`$\overline{(k_i \texttt{ => } \tau_i)_{i \in I}}$`}` **do**
        if $k_0 = k$ **then**
          return τ_0
        end if
      end for
      fail
    default
      fail
end procedure

se verifica $(\tau_1^{++}, \tau_2^{++}) =$ (`[any]`, `[any]`) y $\tau^{[k]} =$ `%{`k `=> any}`. Muy probablemente, sería posible aplicar la traducción TRANSLATE_BIN_OP sin más cambios que generalizar la prueba de correctitud estática de la traducción.

6.2.2. Casts en la evaluación del patrón

Una posibilidad que se podría explorar en el futuro es la de tratar el tipado de patrones de una manera similar a la que se trata a los operadores básicos en el sistema de tipos. Explícitamente, se propone abandonar el subsistema de chequeo-refinamiento de patrones estático y modificar T_MATCH (análogamente T_CASE y T_DEF) a:

$$\frac{\Delta; \Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad matchType(\Gamma, \emptyset, \tau, p) = (\Gamma'', \sigma)}{\Delta; \Gamma \vdash^t p \texttt{ = } e : \sigma \Rightarrow \Gamma' \dagger \Gamma''} \text{ (T\_MATCH)}$$

siendo que $matchType : \text{Map}(Id, T) \times \text{Map}(Id, T) \times T \times P \rightharpoonup \text{Map}(Id, T) \times T$ es una función a nivel de tipos que, pudiendo implementarse como un procedimiento similar a los que fueron presentados, internaliza

las reglas que validan los juicios j. colección de variables en patrones y j. chequeo de patrones bajo contexto de variables.

La regla de traducción sería modificada, de manera similar a los operadores básicos, tomando como punto de partida[1]:

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \widetilde{matchType}(\Gamma, \emptyset, \tau, p) = (\Gamma'', \sigma)}{\Delta;\Gamma \vdash^t p \;=\; e \rightsquigarrow p \;=\; (e \mid \sigma \mathrel{«\!\sim} \tau) : \sigma \Rightarrow \Gamma' \dagger \Gamma''} \text{ (TRANSLATE\_MATCH)}$$

donde

$$(e \mid \sigma \mathrel{«\!\sim} \tau) = \begin{cases} e & \tau = (\tau \leftharpoonup \sigma) \\ (e \mid \tau \sim\!> (\sigma \leftharpoonup \tau)) & \text{si no} \end{cases} \tag{6.1}$$

y $\leftharpoonup$ es análoga al operador de merge ($\leftharpoonup$) pero invirtiendo los roles de $\sim$ y $\leq$, con lo que se tiene la propiedad: Si $\sigma \lesssim \tau$, entonces $\sigma \leq (\tau \leftharpoonup \sigma) \sim \tau$.

Para esclarecer el funcionamiento de la regla TRANSLATE_MATCH, se ilustra con un ejemplo:

$$\frac{[y \mapsto \texttt{any}];\emptyset \vdash^t \texttt{y} : \texttt{any} \Rightarrow \emptyset \qquad \widetilde{matchType}(\emptyset, \emptyset, \texttt{any}, \texttt{\{x,:one\}}) = ([x \mapsto \texttt{any}], \texttt{\{any,:one\}})}{\Delta;\Gamma \vdash^t \texttt{\{x,:one\}} \;=\; \texttt{y} \rightsquigarrow \texttt{\{x,:one\}} \;=\; (\texttt{y} \mid \texttt{any} \sim\!> \texttt{\{any,:one\}}) : \texttt{\{any,:one\}} \Rightarrow [x \mapsto \texttt{any}]}$$

Con el nuevo cast se logra evitar `MatchError` cuando éstos se deben a problemas que únicamente involucran riesgos dinámicos.

Es importante mencionar que los cambios propuestos deben ser analizados cuidadosamente previo a ser integrados, pues deben garantizar la SGG y la correctitud estática de la traducción.

6.2.3. Mejoras en el juicio de chequeo del patrones

Se propone modificar el juicio j. colección de variables en patrones $\Sigma;\Gamma;\tau \vdash^{pc} p \Rightarrow \Gamma;\Sigma'$ adaptando además la regla TPC_PIN:

$$\frac{\Sigma[x] = \sigma \quad \tau \wedge \sigma = \mu}{\Sigma;\Gamma;\tau \vdash^{pc} p \Rightarrow \Gamma;\Sigma[x \mapsto \mu]} \text{ (TPC\_PIN)}$$

Con dicha regla se lograría refinar los tipos del contexto externo, y no solo el de los tipos del nuevo contexto del patrón. Esto requiere de cambios en el juicio j. chequeo-refinamiento de patrones y la regla TCP:

‘ En ese caso T_MATCH luce como:

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \Gamma;\tau \vdash^{pt} e : \sigma \Rightarrow \Gamma'';\Gamma'''}{\Delta;\Gamma \vdash^t p \;=\; e : \sigma \Rightarrow \Gamma''' \dagger \Gamma' \dagger \Gamma''} \text{ (T\_MATCH)}$$

siendo que $\Gamma''' \dagger \Gamma' \dagger \Gamma''$ contiene (por orden de precedencia):

1. En Γ'' las variables exportadas tras el chequeo-refinamiento de p con τ.
2. En Γ' las variables exportadas tras el chequeo de e.
3. En Γ''' las variables del contexto previo al chequeo de e, posiblemente refinadas

Observar que las mejoras que se proponen aquí podrían sumarse a las de la sección anterior modificando de manera acorde la definición de *matchType*.

[1] aunque deberá complejizarse para poder respetar la prueba de correctitud estática de la traducción

6.2.4. Supremo gradual arbitrario con semántica AGT

Como se mencionó en 3.2.3.3, el supremo arbitrario según AGT no se extiende por iteración del caso binario, por lo que su definición se ha omitido del capítulo correspondiente.

Una idea a explorar para la definición del supremo arbitrario (finito), parece ser introduciendo dos "tipos ficticios" $b_{\mathsf{any}}, b^{\mathsf{any}}$ para cada $b \in B$. Luego, se adapta la definición binaria GSup en los casos graduales para reflejar:

1. $\text{GSup}(b, \mathsf{any}, \text{true}) = b^{\mathsf{any}}$ si τ no es maximal
2. $\text{GSup}(b, \mathsf{any}, \text{false}) = b_{\mathsf{any}}$ si τ no es minimal

y los casos estáticos (ahora incluyendo los tipos ficticios)

1. $\text{GSup}(b^{\mathsf{any}}, b', \text{true}) = \text{GSup}(b, b', \text{true})^{\mathsf{any}}$
2. $\text{GSup}(b_{\mathsf{any}}, b', \text{false}) = \text{GSup}(b, b', \text{false})_{\mathsf{any}}$
3. $\text{GSup}(b^{\mathsf{any}}, \mathsf{any}, \text{true}) = \mathsf{any}$
4. $\text{GSup}(b_{\mathsf{any}}, \mathsf{any}, \text{false}) = \mathsf{any}$

Luego, la definición del supremo arbitrario se sigue de iterar GSup (con la modificación de arriba) de izquierda a derecha y finalmente declarar que $b^{\mathsf{any}} = b$ si τ es maximal y any si no; $b_{\mathsf{any}} = b$ si b es minimal y any si no; en una recorrida por toda la estructura del tipo resultado. Por ejemplo, siguiendo retomando el último ejemplo de la Sección 3.2.3.3:

$$\widetilde{\bigvee}\{\texttt{\{integer,any\}}, \texttt{\{float,integer\}}, \texttt{\{any,float\}}\} = \texttt{\{number,number\}}$$

se implementa siguiendo los siguientes pasos:

$\texttt{GSup}(\{\texttt{\{integer,any\}}, \texttt{\{float,integer\}}, \texttt{\{any,float\}}\})$
$\rightarrow \texttt{GSup}(\texttt{GSup}(\{\texttt{\{integer,any\}}, \texttt{\{float,integer\}}), \texttt{\{any,float\}}\})$
$\rightarrow \texttt{GSup}(\{\texttt{\{number,integer}^{\mathsf{any}}\texttt{\}}, \texttt{\{any,float\}}\})$
$\rightarrow \{\texttt{number}^{\mathsf{any}}\texttt{,number}^{\mathsf{any}}\texttt{\}}$
$\rightarrow \texttt{\{number,number\}}$

6.2.5. Formalización de la semántica y la gradual guarantee

Para poder postular (y demostrar) varias de las propiedades formales usuales (*type safety*, *Blame Theorem* y la *Dynamic Gradual Guarantee*, por ejemplo) es necesario definir una semántica de evaluación *small-step*. En general, esta semántica corresponde a un *sistema de transición* definido por una noción de *fase* y una función de *transición entre fases* (ver Capítulo 5 de Harper [2016]).

Ingenuamente, podría definirse la fase en base a tuplas de la forma $\langle \Gamma, e \rangle$ y definir reglas de transición entre fases $\langle \Gamma, e \rangle \rightarrow \langle \Gamma', e' \rangle$ de la manera habitual. Por ejemplo, para las tuplas (binarias, para simplificar):

$$\frac{\langle \Gamma, e_1 \rangle \rightarrow \langle \Gamma', e_1' \rangle}{\langle \Gamma, \{e_1, e_2\} \rangle \rightarrow \langle \Gamma', \{e_1', e_2\} \rangle} \qquad \frac{\langle \Gamma, e_2 \rangle \rightarrow \langle \Gamma', e_2' \rangle}{\langle \Gamma, \{v, e_2\} \rangle \rightarrow \langle \Gamma', \{v, e_2\} \rangle}$$

Notar que esta semántica no refleja correctamente la evaluación de las tuplas en Elixir, ya que e_2 debería comenzar con el mismo contexto Γ que e_1. Parece ser que para lograr lo anterior, sería necesario mantener una estructura de *pila* de contextos y nuevas reglas para el manejo explícito de esta pila.

Referencias

M. Cassola, A. Talagorria, A. Pardo, and M. Viera. A gradual type system for elixir. *Journal of Computer Languages*, 68:101077, 2022. ISSN 2590-1184. https://doi.org/10.1016/j.cola.2021.101077. URL https://www.sciencedirect.com/science/article/pii/S2590118421000551.

G. Castagna, V. Lanvin, T. Petrucciani, and J. G. Siek. Gradual typing: a new perspective. *Proceedings of the ACM on Programming Languages*, 3:1–32, 2019. ISSN 2475-1421.

M. Cimini and J. G. Siek. The gradualizer: A methodology and algorithm for generating gradual type systems. *ACM SIGPLAN Notices*, 51:443–455, 4 2016. ISSN 15232867. 10.1145/2837614.2837632. URL http://dx.doi.org/10.1145/2837614.2837632.

M. Cimini and J. G. Siek. Automatically generating the dynamic semantics of gradually typed languages. *ACM SIGPLAN Notices*, 52, 2017. ISSN 03621340. 10.1145/3093333. URL http://dx.doi.org/10.1145/3009837.3009863.

Elixir team. Typespecs. https://hexdocs.pm/elixir/1.14/typespecs.html, 2021.

Elixir team. Mix. https://hexdocs.pm/mix/1.14/Mix.html, 2022a.

Elixir team. Operators. https://hexdocs.pm/elixir/1.12/operators.html, 2022b.

Elixir team. Alias, require and imports. https://elixir-lang.org/getting-started/alias-require-and-import.html, 2023a.

Elixir team. Macros. https://elixir-lang.org/getting-started/meta/macros.html, 2023b.

A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. *Proceedings-Symposium on Logic in Computer Science*, pages 137–146, 2002. ISSN 10436871. 10.1109/LICS.2002.1029823.

R. Garcia, A. M. Clark, and Éric Tanter. Abstracting gradual typing. *ACM SIGPLAN Notices*, 51:429–442, 4 2016. ISSN 0362-1340. 10.1145/2914770.2837670. URL https://dl.acm.org/doi/abs/10.1145/2914770.2837670.

R. Harper. *Practical foundations for programming languages, second edition*. Cambridge University Press, 1 2016. ISBN 9781316576892. 10.1017/CBO9781316576892.

J. Huffman. Dialyxir. https://github.com/jeremyjh/dialyxir, 2022.

S. Juric. *Elixir in action*. Simon and Schuster, 2019.

S. Jurić. Understanding elixir macros. https://www.theerlangelist.com/article/macros_6, 2014.

T. Lindahl and K. Sagonas. Practical type inference based on success typings. *PPDP'06 - Proceedings of the Eight ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2006:167–178, 2006. 10.1145/1140335.1140356.

B. Pierce. *Types and programming languauges*. The MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

Racket team. Racket. https://racket-lang.org/, 2022.

J. Siek and W. Taha. Gradual typing for functional languages. pages 81–92, 01 2006.

J. Siek and W. Taha. Gradual typing for objects. In *LNPSE,volume 4609*, volume ECOOP 2007 – Object-Oriented Programming, 2007, Volume 4609, pages 2–27. Springer, 2007. ISBN 9783540735885. 10.1007/978-3-540-73589-2_2. URL https://link.springer.com/chapter/10.1007/978-3-540-73589-2_2.

J. G. Siek and P. Wadler. Threesomes, with and without blame. *ACM Sigplan Notices*, 45(1):365–376, 2010.

J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32, pages 274–293. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 5 2015. ISBN 9783939897804. 10.4230/LIPICS.SNAPL.2015.274.

G. Tabone and A. Francalanza. Session types in elixir. *AGERE 2021 - Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, co-located with SPLASH 2021*, pages 12–23, 10 2021. 10.1145/3486601.3486708.

D. Thomas, A. Hunt, and C. Fowler. *Programming Ruby 1.9 & 2.0: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2013.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. *ACM SIGPLAN Notices*, 43:395–406, 1 2008. ISSN 15232867. 10.1145/1328897.1328486. URL https://dl.acm.org/doi/abs/10.1145/1328897.1328486.

S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, pages 117–128, 2010. 10.1145/1863543.1863561.

Apéndice A

Sintaxis

A continuación se especifica el subconjunto de la sintaxis del lenguaje elixir que se incluye en el fragmento de estudio.

Literales:

$$
\begin{array}{rcl}
l & ::= & i \mid f \mid a \\
i & ::= & [0-9]^{+} \\
f & ::= & [0-9]^{+}.[0-9]^{+} \\
c & ::= & \text{“}[a-z, A-Z, 0-9]^{*}\text{”} \\
a & ::= & :[a-z][a-z, A-Z, 0-9]^{*}
\end{array}
$$

Identificadores:

$$
\begin{array}{rcl}
id & ::= & [a-z][a-z, A-Z, 0-9, \_]^{*} \\
x & ::= & id \\
f & ::= & id \\
arity & ::= & [0-9]^{+}
\end{array}
$$

Patrones:

$$
\begin{array}{rcl}
p & ::= & \_ \mid l \mid x \mid \hat{}\, x \\
 & \mid & [\,] \mid\mid [p \mid p] \mid \{p, \ldots, p\} \mid \%\{l \texttt{ => } p, \ldots, l \texttt{ => } p\}
\end{array}
$$

Expresiones:

$$
\begin{array}{rcl}
e & ::= & l \mid x \mid \ominus e \mid e \oplus e \\
& \mid & \texttt{[]} \mid \texttt{[}e\texttt{|}e\texttt{]} \mid \texttt{\{}e,\ldots,e\texttt{\}} \mid \texttt{\%\{}l\texttt{ => }e,\ldots,l\texttt{ => }e\texttt{\}} \\
& \mid & (e;)^{*}e \\
& \mid & \texttt{if } e \texttt{ do } e \texttt{ else } e \texttt{ end} \\
& \mid & \texttt{case } e \texttt{ do } (p \texttt{ => } e)^{+} \texttt{ end} \\
& \mid & \texttt{\&}f\_name/arity \mid f\_name\texttt{(}e,\ldots,e\texttt{)} \mid x\texttt{.(}e,\ldots,e\texttt{)}
\end{array}
$$

Declaraciones de función:

$$
f \quad ::= \quad \texttt{def } f\_name\texttt{(}p,\ldots,p\texttt{) do } e \texttt{ end}
$$

Declaración de módulo:

$$
\begin{array}{rcl}
s & ::= & \texttt{@spec } f\_name\texttt{(}t\texttt{, ..., }t\texttt{) :: } t \\
d & ::= & f \mid s \\
p & ::= & \texttt{defmodule Program do } d\texttt{ , ..., } d \texttt{ end}
\end{array}
$$

Apéndice B

Sistema de tipos estático

B.1. Reglas para el subtipado

$$\frac{}{\texttt{integer} \leq \texttt{number}}\ (\text{ST\_INTEGER}) \qquad \frac{}{\texttt{float} \leq \texttt{number}}\ (\text{ST\_FLOAT})$$

$$\frac{}{a \leq \texttt{atom}}\ (\text{ST\_ATOM}) \qquad \frac{a \in \{\texttt{:true}, \texttt{:false}\}}{a \leq \texttt{boolean}}\ (\text{ST\_BOOL}) \qquad \frac{}{\texttt{boolean} \leq \texttt{atom}}\ (\text{ST\_BOOLEAN})$$

$$\frac{\tau \in B}{\tau \leq \tau}\ (\text{ST\_B\_REFL})$$

$$\frac{}{\texttt{[]} \leq \texttt{[]}}\ (\text{ST\_ELIST\_REFL}) \qquad \frac{}{\texttt{[]} \leq \texttt{[}\tau\texttt{]}}\ (\text{ST\_ELIST}) \qquad \frac{\tau \leq \sigma}{\texttt{[}\tau\texttt{]} \leq \texttt{[}\sigma\texttt{]}}\ (\text{ST\_LIST})$$

$$\frac{(k_{i_1}, ..., k_{i_n}) = (l_1, ..., l_n) \text{ con } \{i_1, ..., i_n\} \subseteq I \qquad \tau_{k_{i_1}} \leq \sigma_{l_1} \quad \cdots \quad \tau_{k_{i_n}} \leq \sigma_{l_n}}{\texttt{\%\{}(\overline{k_i \texttt{ => } \tau_i})_{i \in I}\texttt{\}} \leq \texttt{\%\{}(\overline{l_j \texttt{ => } \sigma_j})_{i \in \{1,...,n\}}\texttt{\}}}\ (\text{ST\_MAP})$$

$$\frac{\tau_i \leq \sigma_i \quad \forall i \in \{1, ..., n\}}{\{\tau_1, ..., \tau_n\} \leq \{\sigma_1, ..., \sigma_n\}}\ (\text{ST\_TUPLE})$$

$$\frac{\tau \leq \sigma \qquad \sigma_i \leq \tau_i \quad \forall i \in \{1, ..., n\}}{(\tau_1, .., \tau_n) \to \tau_0 \leq (\sigma_1, .., \sigma_n) \to \sigma_0}\ (\text{ST\_FUN})$$

B.2. Algoritmo del subtipado

procedure SUBTYPE(τ, σ)
  switch (τ, σ, b) **do**
    case (τ, τ)
      return `:true`
    case (`integer`, `number`) **or** (`float`, `number`)
      return `:true`
    case (`:true`, `boolean`) **or** (`:false`, `boolean`)
      return `:true`
    case (a, `atom`)
      return `:true`
    case (`boolean`, `atom`)
      return `:true`
    case (`[`τ`]`, `[`σ`]`)
      return SUBTYPE(τ, σ)
    case ($\{\tau_1, ..., \tau_n\}$, $\{\sigma_1, ..., \sigma_n\}, b$)
      return SUBTYPE(τ_1, σ_1, b) **and** ... **and** SUBTYPE(τ_n, σ_n, b)
    case (`%{`$\overline{(k_i \Rightarrow \tau_i)_{i\in I}}$`}`, `%{`$\overline{(l_j \Rightarrow \sigma_j)_{i\in J}}$`}`, **true**)
      $isSubtype \leftarrow$ **true**
      for k, σ **in** `%{`$\overline{(k_i \Rightarrow \sigma_i)_{i\in J}}$`}` **do**
        $\tau \leftarrow$ GETS(`%{`$\overline{(k_i \Rightarrow \tau_i)_{i\in I}}$`}`, k)
        if $\tau =$ **fail then**
          return false
        end if
        $isSubtype \leftarrow isSubtype$ **and** SUBTYPE(τ, σ)
      end for
    case $((\tau_1, .., \tau_n) \rightarrow \tau_0, (\sigma_1, .., \sigma_n) \rightarrow \sigma_0, b)$
      return $(\text{SUP}(\tau_1, \sigma_1, \textbf{not } b), .., \text{SUP}(\tau_n, \sigma_n, \textbf{not } b)) \rightarrow \text{SUP}(\tau_0, \sigma_0, b)$
    default
      fail
end procedure

B.3. Algoritmo del supremo/ínfimo

procedure SUP(τ, σ, b)
  switch (τ, σ, b) **do**
    case ($\tau, \sigma,$ **true**) **when** SUBTYPE(τ, σ)
      return σ
    case ($\tau, \sigma,$ **false**) **when** SUBTYPE(τ, σ)
      return τ
    case (`integer`, `float`, **true**) **or** (`float`, `integer`, **true**)
      return `number`
    case ($a_1, a_2,$ **true**) **when** $a_1, a_2 \in$ {`:true`, `:false`}
      return `boolean`
    case ($a_1, a_2,$ **true**)
      return `atom`
    case (`[`τ`]`, `[`σ`]`, b)
      return `[`SUP(τ, σ, b)`]`
    case ({$\tau_1, \ldots, \tau_n$}, {$\sigma_1, \ldots, \sigma_n$}, b)
      return {SUP(τ_1, σ_1, b), ..., SUP(τ_n, σ_n, b)}
    case (`%{`$\overline{(k_i \Rightarrow \tau_i)_{i\in I}}$`}`, `%{`$\overline{(k_i \Rightarrow \sigma_i)_{i\in J}}$`}`, **true**)
      return `%{`$\overline{(k_i \Rightarrow \text{SUP}(\tau_i, \sigma_i, b))_{i\in I\cap J}}$`}`
    case (`%{`$\overline{(k_i \Rightarrow \tau_i)_{i\in I}}$`}`, `%{`$\overline{(k_i \Rightarrow \sigma_i)_{i\in J}}$`}`, **false**)
      return `%{`$\overline{(k_i \Rightarrow \text{SUP}(\tau_i, \sigma_i, b))_{i\in I\cap J}}$, $\overline{(k_i \Rightarrow \tau_i)_{i\in I\setminus J}}$, $\overline{(k_i \Rightarrow \sigma_i)_{i\in J\setminus I}}$`}`
    case ($(\tau_1, .., \tau_n) \to \tau_0, (\sigma_1, .., \sigma_n) \to \sigma_0, b$)
      return (SUP($\tau_1, \sigma_1,$ **not** b), .., SUP($\tau_n, \sigma_n,$ **not** b)) $\to$ SUP(τ_0, σ_0, b)
    default
      fail
end procedure

B.4. Definición de UniOpType y BinOpType

A continuación se da la definición completa de las funciones *uniOpType* y *binOpType* auxiliares a las reglas T_UNI_OP y T_BIN_OP respectivamente.

```
procedure UniOpType(⊖, τ)
    switch (⊖, τ) do
        case (-, integer)
            return integer
        case (-, float)
            return float
        case (-, number)
            return number
        case (abs, integer)
            return integer
        case (abs, float)
            return float
        case (abs, number)
            return number
        case (not, :true)
            return :false
        case (not, :false)
            return :true
        case (not, boolean)
            return boolean
        default
            fail
end procedure
```

```
procedure BinOpType(⊖, τ, σ)
    switch ⊕ do
        case and
            return ConjunctionType(τ, σ)
        case or
            return DisjunctionType(τ, σ)
        case +, -, *
            return SumType(τ, σ)
        case /
            return DivType(τ, σ)
        case div, rem
            return IntegerDivType(τ, σ)
        case <>
            return ConcatType(τ, σ)
        case ==, !=
            return EqualityType(τ, σ)
        case <, >, <=, >=, ===, !==
            return IdentityType(τ, σ)
        default
            fail
end procedure
```

```
procedure ConjunctionType(τ, σ)
    switch (τ, σ) do
        case (:true, boolean)
            return integer
        case (:false, σ) when σ ≤ boolean
            return :false
        case (τ, :false) when τ ≤ boolean
            return :false
        case (τ, σ) when τ, σ ≤ boolean
            return boolean
        default
            fail
end procedure
```

```
procedure DisjunctionType(τ, σ)
    switch (τ, σ) do
        case (:true, boolean)
            return integer
        case (:true, σ) when σ ≤ boolean
            return :true
        case (τ, :true) when τ ≤ boolean
            return :true
        case (τ, σ) when τ, σ ≤ boolean
            return boolean
        default
            fail
end procedure
```

```
procedure SumType(τ, σ)
    switch (τ, σ) do
        case (integer, integer)
            return integer
        case (τ, float) when τ ≤ number
            return float
        case (float, σ) when σ ≤ number
            return float
        case (τ, σ) when τ, and σ ≤ number
            return number
        default
            fail
end procedure

procedure MaxType(τ, σ)
    switch (τ, σ) do
        case (integer, integer)
            return integer
        case (float, float)
            return float
        case (τ, σ) when τ, and σ ≤ number
            return number
        default
            fail
end procedure

procedure ConcatType(τ, σ)
    switch (τ, σ) do
        case (string, string)
            return string
        default
            fail
end procedure

procedure DivType(τ, σ)
    switch (τ, σ) do
        case (τ, σ) when τ, σ ≤ number
            return float
        default
            fail
end procedure

procedure IntegerDivType(τ, σ)
    switch (τ, σ) do
        case (integer, integer)
            return integer
        default
            fail
end procedure

procedure EqualityType(τ, σ)
    switch (τ, σ) do
        case (string, string)
            return :true
        default
            fail
end procedure

procedure IdentityType(τ, σ)
    switch (τ, σ) do
        case (τ, σ) when τ, σ ≤ number
            return boolean
        default
            fail
end procedure
```

B.5. Definición de MaxArgType

Las siguientes definiciones indican, respectivamente, los tipos $\tau^{\ominus}$ y $(\tau_1^{\oplus}, \tau_2^{\oplus})$ que realizan la propiedad del argumento máximo de la Sección 3.1.8.3.1.

```
procedure MaxArgUniOpType(⊖)
    switch ⊖ do
        case -, abs
            return number
        case not
            return boolean
        default
            fail
end procedure
```

```
procedure MaxArgBinOpType(⊕)
    switch ⊕ do
        case and
            return (boolean, boolean)
        case or
            return (boolean, boolean)
        case +, -, *, /
            return (number, number)
        case div, rem
            return (integer, integer)
        case <>, ==
            return (string, string)
        case ==, !=
            return (atom, atom)
        case <, >, <=, >=, ===, !==
            return (number, number)
        default
            fail
end procedure
```

B.6. Reglas para el chequeo de un programa

La regla de chequeo es:

$$\frac{\vdash^{c} m \Rightarrow \Delta \qquad \Delta \vdash^{ts} m}{\vdash^{t} m}\ \text{(T\_PROG)}$$

y las reglas para el chequeo con especificaciones son:

$$\frac{\Delta \vdash^{t} d_1 \quad \ldots \quad \Delta \vdash^{t} d_n}{\Delta \vdash^{ts} \texttt{defmodule Program do } d_1\texttt{, ..., } d_n \texttt{ end}}\ \text{(TS\_PROG)}$$

$$\frac{}{\Delta \vdash^{t} s}\ \text{(T\_SPEC)} \qquad \frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \emptyset; \emptyset; \tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset; \Gamma_{n-1}; \tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta; \Gamma_n \vdash^{t} e : \sigma \Rightarrow \Gamma' \quad \sigma \leq \tau_0\end{array}}{\Delta \vdash^{t} \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end}}\ \text{(T\_DEF)}$$

Finalmente las reglas para la colección de especificaciones son:

$$\frac{\emptyset \vdash^{c} d_1 \Rightarrow \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash^{c} d_n \Rightarrow \Delta_n}{\vdash^{c} \texttt{defmodule Program do } d_1\texttt{, ..., } d_n \texttt{ end} \Rightarrow \Delta_n}\ \text{(COLLECT\_PROG)}$$

$$\frac{}{\Delta \vdash^{c} f \Rightarrow \Delta}\ \text{(COLLECT\_DEF)}$$

$$\frac{(f\_name, n) \notin \text{dom}(\Delta)}{\Delta \vdash^{c} \texttt{@spec } f\_name(\tau_1, \ldots, \tau_n) \texttt{ :: } \tau_0 \Rightarrow \Delta[(f\_name, n) \mapsto ((\tau_1, \ldots, \tau_n), \tau_0)]}\ \text{(ECOLLECT\_SPEC)}$$

B.7. Reglas para el chequeo de expresiones

$$\frac{}{\Delta;\Gamma \vdash^t l : type(l) \Rightarrow \emptyset}\ \text{(T\_LIT)} \qquad \frac{\Gamma[x] = \tau}{\Delta;\Gamma \vdash^t x : \tau \Rightarrow \emptyset}\ \text{(T\_VAR)}$$

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad uniOpType(\ominus, \tau) = \sigma}{\Delta;\Gamma \vdash^t \ominus e : \sigma \Rightarrow \Gamma'}\ \text{(T\_UNI\_OP)}$$

$$\frac{\Delta;\Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \vdash^t e_2 : \tau_2 \Rightarrow \Gamma_2 \qquad binOpType(\oplus, \tau_1, \tau_2) = \sigma}{\Delta;\Gamma \vdash^t e_1 \oplus e_2 : \sigma \Rightarrow \Gamma_1 \dagger \Gamma_2}\ \text{(T\_BIN\_OP)}$$

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \Gamma;\tau \vdash^{pt} e : \sigma \Rightarrow \Gamma''}{\Delta;\Gamma \vdash^t p \ = \ e : \sigma \Rightarrow \Gamma' \dagger \Gamma''}\ \text{(T\_MATCH)}$$

$$\frac{}{\Delta;\Gamma \vdash^t \texttt{[ ]} : \texttt{[ ]} \Rightarrow \emptyset}\ \text{(T\_ELIST)} \qquad \frac{\Delta;\Gamma \vdash^t e_i : \tau_i \Rightarrow \Gamma_i \quad i = 1,2 \qquad [\tau_1] \vee \tau_2 = \sigma}{\Delta;\Gamma \vdash^t [e_1 \,|\, e_2] : \sigma \Rightarrow \Gamma_1 \dagger \Gamma_2}\ \text{(T\_CONS)}$$

$$\frac{\Delta;\Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \cdots \qquad \Delta;\Gamma \vdash^t e_n : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^t \{e_1, \ldots, e_n\} : \{\tau_1, \ldots, \tau_n\} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n}\ \text{(T\_TUPLE)}$$

$$\frac{\Delta;\Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Delta;\Gamma \vdash^t e_n : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^t \texttt{\%\{}(\overline{k_i \texttt{ => } e_i})_{i\in[n]}\texttt{\}} : \texttt{\%\{}(\overline{k_i \texttt{ => } \tau_i})_{i\in[n]}\texttt{\}} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n}\ \text{(T\_MAP)}$$

$$\frac{\Delta;\Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \dagger \Gamma_1 \vdash^t e_2 : \tau_2 \Rightarrow \Gamma_2}{\Delta;\Gamma \vdash^t e_1 \texttt{;} e_2 : \tau_2 \Rightarrow \Gamma_1 \dagger \Gamma_2}\ \text{(T\_SEQ)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \quad \tau \leq \texttt{boolean} \\ \Delta;\Gamma \dagger \Gamma' \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \quad \Delta;\Gamma \dagger \Gamma' \vdash^t e_2 : \tau_2 \Rightarrow \Gamma_2 \quad \tau_1 \vee \tau_2 = \sigma\end{array}}{\Delta;\Gamma \vdash^t \texttt{if } e \texttt{ do } e_1 \texttt{ else } e_2 \texttt{ end} : \sigma \Rightarrow \Gamma'}\ \text{(T\_IFE)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \\ \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_1 \vdash^t e_1 : \tau_1 \Rightarrow \Gamma'_1 \quad \cdots \quad \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_n \vdash^t e_n : \tau_n \Rightarrow \Gamma'_n \\ \sigma = \bigvee_{i=1}^{n} \tau_i\end{array}}{\Delta;\Gamma \vdash^t \texttt{if } e_1 \texttt{ do } e_2 \texttt{ else } e_3 \texttt{ end} : \sigma \Rightarrow \Gamma'}\ \text{(T\_CASE)}$$

$$\frac{\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0)}{\Delta;\Gamma \vdash^t \texttt{\&}f\_name/n : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \emptyset}\ \text{(T\_ANON)}$$

$$\text{(T\_DCALL)}\ \frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \Delta;\Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots\ \Delta;\Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n \\ \sigma_1 \leq \tau_1 \ \cdots\ \sigma_n \leq \tau_n\end{array}}{\Delta;\Gamma \vdash^t f\_name(e_1, \ldots, e_n) : \tau_0 \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \qquad \frac{\begin{array}{c}\Delta;\Gamma \vdash^t e : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \Gamma_0 \\ \Delta;\Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots\ \Delta;\Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n \\ \sigma_1 \leq \tau_1 \ \cdots\ \sigma_n \leq \tau_n\end{array}}{\Delta;\Gamma \vdash^t e.(e_1, \ldots, e_n) : \tau_0 \Rightarrow \Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n}\ \text{(T\_ACALL)}$$

B.8. Reglas del chequeo-refinamiento de patrones

La regla para el chequeo-refinamiento de patrones es:

$$\frac{\Sigma; \emptyset; \tau \vdash^{pc} p \Rightarrow \Gamma \qquad \Sigma; \Gamma; \tau \vdash^{ptv} p : \sigma}{\Sigma; \tau \vdash^{pt} p : \sigma \Rightarrow \Gamma} \text{(TCP)}$$

B.8.1. Reglas de colección de variables

$$\frac{\text{ty}(l) \leq \tau}{\Sigma; \Gamma; \tau \vdash^{pc} l \Rightarrow \Gamma} \text{(TPC\_LIT)} \qquad \frac{\Sigma[x] = \sigma \quad \tau \wedge \sigma = \mu}{\Sigma; \Gamma; \tau \vdash^{pc} \hat{}x \Rightarrow \Gamma} \text{(TPC\_PIN)} \qquad \frac{}{\Sigma; \Gamma; \tau \vdash^{pc} \_ \Rightarrow \Gamma} \text{(TPC\_WILD)}$$

$$\frac{x \notin \Gamma}{\Sigma; \Gamma; \tau \vdash^{pc} x \Rightarrow \Gamma[x \mapsto \tau]} \text{(TPC\_VARE)} \qquad \frac{\Gamma[x] = \sigma \quad \tau \wedge \sigma = \mu}{\Sigma; \Gamma; \tau \vdash^{pc} x \Rightarrow \Gamma[x \mapsto \mu]} \text{(TPC\_VARN)}$$

$$\frac{\Sigma; \Gamma; \tau \vdash^{pc} p_1 \Rightarrow \Gamma' \quad \Sigma; \Gamma'; [\tau] \vdash^{pc} p_2 \Rightarrow \Gamma''}{\Sigma; \Gamma; [\tau] \vdash^{pc} [p_1 \,|\, p_2] \Rightarrow \Gamma''} \text{(TPC\_CONS)} \qquad \frac{[\,] \leq \tau}{\Sigma; \Gamma; \tau \vdash^{pc} [\,] \Rightarrow \Gamma} \text{(TPC\_ELIST)}$$

$$\text{(TPC\_TUPLE)} \ \frac{\Sigma; \Gamma; \tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Sigma; \Gamma_{n-1}; \tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n}{\Sigma; \Gamma; \{\tau_1, \ldots, \tau_n\} \vdash^{pc} \{p_1, \ldots, p_n\} \Rightarrow \Gamma_n}$$

$$\frac{\begin{array}{c}(k_{i_1}, .., k_{i_n}) = (l_1, .., l_n) \text{ con } \{i_1, .., i_n\} \subseteq I \\ \Sigma; \Gamma; \tau_{k_{i_1}} \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Sigma; \Gamma_{n-1}; \tau_{k_{i_n}} \vdash^{pc} p_n \Rightarrow \Gamma_n\end{array}}{\Sigma; \Gamma; \%\{(\overline{k_i \Rightarrow \tau_i})_{i \in I}\} \vdash^{pc} \%\{(\overline{l_i \Rightarrow p_i})_{i \in \{1,\ldots,n\}}\} \Rightarrow \Gamma_n} \text{(TPC\_MAP)}$$

B.8.2. Reglas para el juicio de refinamiento en patrones

$$\frac{}{\Sigma; \Gamma; \tau \vdash^{ptv} l : type(l)} \text{(TPV\_LIT)} \qquad \frac{\Sigma[x] = \sigma}{\Sigma; \Gamma; \tau \vdash^{ptv} \hat{}x : \sigma} \text{(TPV\_PIN)} \qquad \frac{}{\Sigma; \Gamma; \tau \vdash^{pc} \_ \Rightarrow \tau} \text{(TPV\_WILD)}$$

$$\frac{\Gamma[x] = \sigma}{\Sigma; \Gamma; \tau \vdash^{ptv} x : \sigma} \text{(TPV\_VAR)} \qquad \frac{}{\Sigma; \Gamma; \tau \vdash^{ptv} [\,] : [\,]} \text{(TPV\_ELIST)}$$

$$\frac{\Sigma; \Gamma; \tau \vdash^{ptv} p : \sigma_1 \quad \Sigma; \Gamma; [\tau] \vdash^{ptv} p_2 : \sigma_2 \quad [\sigma_1] \vee \sigma_2 = \sigma}{\Sigma; \Gamma; [\tau] \vdash^{ptv} [p_1 \,|\, p_2] : \sigma} \text{(TPV\_CONS)}$$

$$\frac{\Sigma; \Gamma; \tau_1 \vdash^{ptv} p_1 : \sigma_1 \quad \cdots \quad \Sigma; \Gamma; \tau_n \vdash^{ptv} p_n : \sigma_n}{\Sigma; \Gamma; \{\tau_1, \ldots, \tau_n\} \vdash^{ptv} \{p_1, \ldots, p_n\} : \{\sigma_1, \ldots, \sigma_n\}} \text{(TPV\_TUPLE)}$$

$$\frac{\begin{array}{c}(k_{i_1}, .., k_{i_n}) = (l_1, .., l_n) \text{ con } \{i_1, .., i_n\} \subseteq I \\ \Sigma; \Gamma; \tau_{k_{i_1}} \vdash^{ptv} p_1 : \sigma_{i_1} \quad \cdots \quad \Sigma; \Gamma; \tau_{k_{i_n}} \vdash^{ptv} p_n : \sigma_{i_n} \\ \sigma_i = \tau_i \quad \forall i \in I \setminus \{i_1, .., i_n\}\end{array}}{\Sigma; \Gamma; \%\{(\overline{k_i \Rightarrow \tau_i})_{i \in I}\} \vdash^{ptv} \%\{(\overline{l_i \Rightarrow p_i})_{i \in \{1,\ldots,n\}}\} : \%\{(\overline{k_i \Rightarrow \sigma_i})_{i \in I}\}} \text{(TPV\_MAP)}$$

B.9. Algoritmo de refinamiento de patrones

procedure REFINE(Σ, Γ, τ, p)
  switch (τ, p) **do**
    case (τ, l)
      return ty(l)
    case (τ, _)
      return τ
    case (τ, ^x)
      return $\Sigma[x]$
    case (τ, x)
      return $\Gamma[x]$
    case (τ, [ ])
      return [ ]
    case ([τ], [p_1 | p_2])
      $\sigma_1 \leftarrow$ SUP($\Sigma, \Gamma, \tau, p_1$)
      $\sigma_2 \leftarrow$ SUP(Σ, Γ, [τ], p_2)
      return SUP([σ_1], σ_2, **true**)
    case ($\{\tau_1, \ldots, \tau_n\}$, $\{p_1, \ldots, p_n\}$)
      return {REFINE($\Sigma, \Gamma, \tau_1, p_1$), ..., REFINE($\Sigma, \Gamma, \tau_n, p_n$)}
    case (%{$\overline{(k_i \Rightarrow \tau_i)}_{i \in I}$}, %{$\overline{(l_j \Rightarrow p_j)}_{i \in J}$})
      $ks \leftarrow$ []
      $\sigma s \leftarrow$ []
      for k, τ **in** %{$\overline{(k_i \Rightarrow \tau_i)}_{i \in I}$} **do**
        $\sigma \leftarrow \tau$
        $p \leftarrow$ GETS(%{$\overline{(k_i \Rightarrow \tau_i)}_{i \in I}$}, k)
        if $p \neq$ **fail then**
          $\sigma \leftarrow$ REFINE(Σ, Γ, τ, p)
        end if
        $ks \leftarrow$ APPEND(ks, k)
        $\sigma s \leftarrow$ APPEND($\sigma s, \sigma$)
        return %{$\overline{(\text{GETS}(ks, i) \Rightarrow \text{GETS}(\sigma s, i))}_{i \in \{1,\ldots,n\}}$}
      end for
    default
      fail
end procedure

B.10. Relación del chequeo de patrones con *Ocurrence Typing*

Es posible trazar vínculos entre la metodología aplicada en este trabajo para establecer el subsistema de chequeo-refinamiento de patrones y la disciplina denominada *Ocurrence Typing* por sus creadores en Tobin-Hochstadt and Felleisen [2008,0] para la adopción de sistemas de tipos en lenguajes dinámicamente tipados (y particularmente para el lenguaje *Racket*).

En los trabajos mencionados, se exhibe un sistema de tipos que permite, de manera muy general, extraer información estática a partir de la posición relativa de una expresión en el flujo del programa en presencia de testeos de tipos. Un ejemplo básico que ilustra la idea fundamental (escrito en Elixir completo):

```
@spec is_even(number) :: boolean
def is_even(x) do
    if is_integer(x) do
        mod(x, 2) == 0
    else
        false
    end
end
```

Si se aplican los principios de Ocurrence Typing, la función `is_even` pasaría el chequeo porque la presencia del predicado `is_integer(x)` permite enriquecer el contexto Γ con x de tipo `integer` en la primera rama y `float` en la segunda.

El vínculo con el juicio de chequeo de patrones aquí presentado se base en el hecho que cada instancia de pattern match admite una reescritura equivalente utilizando testeos de tipos y destructores, la cual pasaría el chequeo en aquel sistema - y produciendo la misma información estática- al incorporar los principios del Ocurrence Typing.

Por ejemplo, para el caso

```
{x, x, y} = e
...
```

corresponde el juicio

$$\emptyset; \texttt{\{number,integer,number\}} \vdash^{pt} \texttt{\{x,x,1\}} : \texttt{\{integer,integer,integer\}} \Rightarrow [x \mapsto \texttt{integer}]$$

El resultado de la reescritura usando testeos de tipo y destructores es

```
if is_tuple(e) do
    if tuple_size(e) === 3 do
        x = elem(e, 0)
        x = elem(e, 1)
        y = elem(e, 2)
        if is_number(x) do
            if is_integer(x) do
                if is_integer(y) do
                    if y == 1 do
                        {x, x, y}
```

```
                            ...
                        end
                    end
                end
            end
        end
    end
    raise MatchError
```

A modo de conclusión: sería posible explicar el diseño de las reglas que definen el juicio de chequeo de patrones como un caso particular[1] de los refinamientos que se obtienen siguiendo los principios básicos de Ocurrence Typing desarrollada en Tobin-Hochstadt and Felleisen [2010] al tomar en cuenta la reescritura de patrones con testeos de tipos y destructores delineada arriba.

Parte del interés del vínculo descrito se plantea en la sección de trabajo futuro, en relación a la búsqueda de principios que guíen una reescritura con casts para la evaluación de patrones en el contexto gradual.

[1] Pues el sistema que se presenta allí logra extender el análisis de refinamiento al nivel de una expresión o función arbitraria

Apéndice C

Sistema de tipos gradual

C.1. Reglas para la precisión

C.1.1. Reglas para precisión entre expresiones

$$\frac{}{\tau \lll \texttt{any}} \text{ (P\_ANY)} \qquad \frac{}{\tau \lll \tau} \text{ (P\_REFL)}$$

$$\frac{\tau \lll \sigma}{[\tau] \lll [\sigma]} \text{ (P\_LIST)} \qquad \frac{\tau_i \lll \sigma_i \quad \forall i \in \{1,...,n\}}{\{\tau_1, ..., \tau_n\} \lll \{\sigma_1, ..., \sigma_n\}} \text{ (P\_TUPLE)}$$

$$\frac{\tau_i \lll \sigma_i \quad \forall i \in \{1,...,n\}}{\texttt{\%\{}(\overline{k_i \texttt{ => } \tau_i})_{i \in I}\texttt{\}} \lll \texttt{\%\{}(\overline{k_i \texttt{ => } \sigma_i})_{i \in I}\texttt{\}}} \text{ (P\_MAP)} \qquad \frac{\tau_i \lll \sigma_i \quad \forall i \in \{0,...,n\}}{(\tau_1, .., \tau_n) \to \tau_0 \lll (\sigma_1, .., \sigma_n) \to \sigma_0} \text{ (P\_FUN)}$$

C.1.2. Reglas para precisión entre programas

$$\frac{\begin{array}{c}\{1,...,n\} \subseteq \{i_1,...,i_m\} \quad d_1 \lll d'_1 \quad \cdots \quad d_n \lll d'_n \\ \{d_{i_1}, ..., d_{i_m}\} \setminus \{d_1, ..., d_n\} = \{s_{i_1}, ..., s_k\} \\ m = \texttt{defmodule Program do } d_{i_1}\texttt{, ..., } d_{i_m} \texttt{ end} \\ m' = \texttt{defmodule Program do } d'_1\texttt{, ..., } d'_n \texttt{ end}\end{array}}{m \lll m'} \text{ (P\_PROG)}$$

$$\frac{}{\texttt{def } f\_name(p_1, ..., p_n) \texttt{ do } e \texttt{ end} \lll \texttt{def } f\_name(p_1, ..., p_n) \texttt{ do } e \texttt{ end}} \text{ (P\_DEF)}$$

$$\frac{\tau_i \lll \sigma_i \quad \forall i \in \{0,...,n\}}{\texttt{@spec } f\_name(\tau_1, ..., \tau_n) \texttt{ :: } \tau_0 \lll \texttt{@spec } f\_name(\sigma_1, ..., \sigma_n) \texttt{ :: } \sigma_0} \text{ (P\_SPEC)}$$

C.2. Reglas para el subtipado

C.2.1. Reglas para el subtipado gradual

En el contexto gradual, la definición del subtipado se extiende por la única regla

$$\frac{}{\texttt{any} \lesssim \texttt{any}}\ \text{(ST\_ANY)}$$

C.2.2. Reglas para el subtipado consistente

$$\frac{}{\texttt{any} \lesssim \texttt{any}}\ \text{(SC\_ANY\_REFL)} \qquad \frac{}{\texttt{any} \lesssim \tau}\ \text{(SC\_ANY\_L)} \qquad \frac{}{\tau \lesssim \texttt{any}}\ \text{(SC\_ANY\_R)}$$

$$\frac{}{\texttt{integer} \lesssim \texttt{number}}\ \text{(SC\_INTEGER)} \qquad \frac{}{\texttt{float} \lesssim \texttt{number}}\ \text{(SC\_FLOAT)}$$

$$\frac{}{a \lesssim \texttt{atom}}\ \text{(SC\_ATOM)} \qquad \frac{a \in \{\texttt{:true}, \texttt{:false}\}}{a \lesssim \texttt{boolean}}\ \text{(SC\_BOOL)} \qquad \frac{}{\texttt{boolean} \lesssim \texttt{atom}}\ \text{(SC\_BOOLEAN)}$$

$$\frac{\tau \in B}{\tau \lesssim \tau}\ \text{(SC\_B\_REFL)}$$

$$\frac{}{[\,] \lesssim [\,]}\ \text{(SC\_ELIST\_REFL)} \qquad \frac{}{[\,] \lesssim [\tau]}\ \text{(SC\_ELIST)} \qquad \frac{\tau \lesssim \sigma}{[\tau] \lesssim [\sigma]}\ \text{(SC\_LIST)}$$

$$\frac{\begin{array}{c}(k_{i_1}, ..., k_{i_n}) = (l_1, ..., l_n) \text{ con } \{i_1, ..., i_n\} \subseteq I \\ \tau_{k_{i_1}} \lesssim \sigma_{l_1} \quad \cdots \quad \tau_{k_{i_n}} \lesssim \sigma_{l_n}\end{array}}{\%\{(\overline{k_i \Rightarrow \tau_i})_{i \in I}\} \lesssim \%\{(\overline{l_j \Rightarrow \sigma_j})_{i \in \{1,...,n\}}\}}\ \text{(SC\_MAP)}$$

$$\frac{\tau_i \lesssim \sigma_i \quad \forall i \in \{1, ..., n\}}{\{\tau_1, ..., \tau_n\} \lesssim \{\sigma_1, ..., \sigma_n\}}\ \text{(SC\_TUPLE)}$$

$$\frac{\begin{array}{c}\tau \lesssim \sigma \\ \sigma_i \lesssim \tau_i \quad \forall i \in \{1, ..., n\}\end{array}}{(\tau_1, .., \tau_n) \to \tau_0 \lesssim (\sigma_1, .., \sigma_n) \to \sigma_0}\ \text{(SC\_FUN)}$$

C.3. Algoritmo del supremo/ínfimo binario AGT

procedure GSup(τ, σ, b)
  switch (τ, σ, b) **do**
    case (τ, any, **true**) **when** $\tau \in B$ **and** Maximal(τ)
      return τ
    case (τ, any, **true**) **when** $\tau \in B$ **and not** Maximal(τ)
      return any
    case (τ, any, **false**) **when** $\tau \in B$ **and** Minimal(τ)
      return τ
    case (τ, any, **false**) **when** $\tau \in B$ **and not** Minimal(τ)
      return any
    case ([ ], any, **true**)
      return any
    case ([ ], any, **false**)
      return [ ]
    case ([τ], any, **true**)
      return [GSup(any, τ)]
    case ([τ], any, **false**)
      return [any]
    case ({τ_1 , ... , τ_n}, any, b)
      return {GSup(τ_1, any, b) , ... , GSup(τ_n, any, b)}
    case (%{$\overline{(k_i \Rightarrow \tau_i)_{i\in I}}$}, any, b)
      return %{$\overline{(k_i \Rightarrow \text{GSup(any}, \tau_i, b))_{i\in I}}$}
    case ((τ_1, .., τ_n) $\rightarrow$ τ_0, any, b)
      $nb \leftarrow$ **not** b
      return (GSup(τ_1, any, nb), .., GSup(τ_n, any, nb)) $\rightarrow$ GSup(τ_0, any$_0$, b)
    case (any, τ, b)
      return GSup(τ, any, b)
    $\cdots$ // el resto de casos es igual al de Sup
  end procedure

estando Maximal y Minimal definidos como

procedure Minimal(τ)
  switch τ **do**
    case a **when** IsAtom(a)
      return true
    case `string`
      return true
    case `integer`
      return true
    case `float`
      return true
    default
      false
end procedure

procedure Maximal(τ)
  switch τ **do**
    case `atom`
      return true
    case `string`
      return true
    case `number`
      return true
    default
      false
end procedure

C.4. Reglas para el subtipado precisión

Las reglas se definen en conjunto para las dos relaciones $\prec^+$ y $\prec^-$, según el parámetro $p \in \{+, -\}$.

$$\frac{}{\texttt{any} \prec^p \texttt{any}}\ \text{(SP\_ANY\_REFL)} \qquad \frac{}{\texttt{any} \prec^+ \tau}\ \text{(SP\_ANY\_L)} \qquad \frac{}{\tau \prec^- \texttt{any}}\ \text{(SP\_ANY\_R)}$$

$$\frac{}{\texttt{integer} \prec^p \texttt{number}}\ \text{(SP\_INTEGER)} \qquad \frac{}{\texttt{float} \prec^p \texttt{number}}\ \text{(SP\_FLOAT)}$$

$$\frac{}{a \prec^p \texttt{atom}}\ \text{(SP\_ATOM)} \qquad \frac{a \in \{\texttt{:true}, \texttt{:false}\}}{a \prec^p \texttt{boolean}}\ \text{(SP\_BOOL)} \qquad \frac{}{\texttt{boolean} \prec^p \texttt{atom}}\ \text{(SP\_BOOLEAN)}$$

$$\frac{\tau \in B}{\tau \prec^p \tau}\ \text{(SP\_B\_REFL)}$$

$$\frac{}{\texttt{[]} \prec^p \texttt{[]}}\ \text{(SP\_ELIST\_REFL)} \qquad \frac{}{\texttt{[]} \prec^p \texttt{[}\tau\texttt{]}}\ \text{(SP\_ELIST)} \qquad \frac{\tau \leq \sigma}{\texttt{[}\tau\texttt{]} \prec^p \texttt{[}\sigma\texttt{]}}\ \text{(SP\_LIST)}$$

$$\frac{(k_{i_1}, ..., k_{i_n}) = (l_1, ..., l_n) \text{ con } \{i_1, ..., i_n\} \subseteq I \qquad \tau_{k_{i_1}} \prec^p \sigma_{l_1} \quad \cdots \quad \tau_{k_{i_n}} \prec^p \sigma_{l_n}}{\texttt{\%\{}(\overline{k_i \texttt{ => } \tau_i})_{i \in I}\texttt{\}} \prec^p \texttt{\%\{}(\overline{l_j \texttt{ => } \sigma_j})_{i \in \{1,...,n\}}\texttt{\}}}\ \text{(SP\_MAP)}$$

$$\frac{\tau_i \prec^p \sigma_i \quad \forall i \in \{1, ..., n\}}{\{\tau_1, ..., \tau_n\} \prec^p \{\sigma_1, ..., \sigma_n\}}\ \text{(SP\_TUPLE)}$$

$$\frac{\tau \prec^p \sigma \qquad \sigma_i \prec^{-p} \tau_i \quad \forall i \in \{1, ..., n\}}{(\tau_1, .., \tau_n) \to \tau_0 \prec^p (\sigma_1, .., \sigma_n) \to \sigma_0}\ \text{(SP\_FUN)}$$

C.5. Gradualización de UniOpType y BinOpType

A continuación se definen los procedimientos que definen las funciones $\widetilde{uniOpType}$ y $\widetilde{binOpType}$:

```
procedure GUniOpType(⊖, τ)
    switch (⊖, τ) do
        case (-, any)
            return any
        case (abs, any)
            return any
        case (not, any)
            return any
        default
            UniOpType(⊖, τ)
end procedure
```

```
procedure GBinOpType(⊖, τ, σ)
    switch ⊕ do
        case and
            return GConjunctionType(τ, σ)
        case or
            return GDisjunctionType(τ, σ)
        case +, -, *
            return GSumType(τ, σ)
        case /
            return GDivType(τ, σ)
        case div, rem
            return GIntegerDivType(τ, σ)
        case ==, !=
            return GEqualityType(τ, σ)
        case <, >, <=, >=, ===, !==
            return GIdentityType(τ, σ)
        default
            fail
end procedure
```

```
procedure GConjunctionType(τ, σ)
    switch (τ, σ) do
        case (:false, any)
            return :false
        case (any, :false)
            return :false
        case (τ, any) when τ ≤ boolean
            return any
        case (any, σ) when σ ≤ boolean
            return any
        case (any, any)
            return any
        default
            ConjunctionType(τ, σ)
end procedure
```

```
procedure GDisjunctionType(τ, σ)
    switch (τ, σ) do
        case (:true, any)
            return :true
        case (any, :true)
            return :true
        case (τ, any) when τ ≤ boolean
            return any
        case (any, σ) when σ ≤ boolean
            return any
        case (any, any)
            return any
        default
            DisjunctionType(τ, σ)
end procedure
```

```
procedure GSumType(τ, σ)
    switch (τ, σ) do
        case (float, any)
            return float
        case (any, float)
            return float
        case (τ, any) when τ ≤ number
            return any
        case (any, σ) when σ ≤ number
            return any
        default
            SumType(τ, σ)
end procedure
```

```
procedure GMaxType(τ, σ)
    switch (τ, σ) do
        case (τ, any) when τ ≤ number
            return any
        case (any, σ) when σ ≤ number
            return any
        default
            MaxType(τ, σ)
end procedure
```

```
procedure GConcatType(τ, σ)
    switch (τ, σ) do
        case (τ, any) when τ ≤ string
            return string
        case (any, σ) when σ ≤ string
            return string
        default
            StringType(τ, σ)
end procedure
```

```
procedure GDivType(τ, σ)
    switch (τ, σ) do
        case (τ, any) when τ ≤ number
            return float
        case (any, σ) when σ ≤ number
            return float
        default
            DivType(τ, σ)
end procedure
```

```
procedure GIntegerDivType(τ, σ)
    switch (τ, σ) do
        case (any, integer)
            return integer
        case (integer, any)
            return integer
        default
            IntegerType(τ, σ)
end procedure
```

```
procedure GEqualityType(τ, σ)
    switch (τ, σ) do
        case (string, any)
            return boolean
        case (any, string)
            return boolean
        default
            EqualityType(τ, σ)
end procedure
```

```
procedure GIdentityType(τ, σ)
    switch (τ, σ) do
        case (τ, any) when τ ≤ number
            return boolean
        case (any, σ) when σ ≤ number
            return boolean
        default
            IdentityType(τ, σ)
end procedure
```

C.6. Reglas para el chequeo de un programa

El primer cambio está en que se debe añadir la regla T_DEF_ANY para el chequeo de definiciones no especificadas:

$$\frac{\begin{array}{c}(f\_name, n) \notin \mathrm{dom}(\Delta) \\ \emptyset;\emptyset;\mathsf{any} \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset;\Gamma_{n-1};\mathsf{any} \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta;\Gamma_n \vdash^{t} e : \sigma \Rightarrow \Gamma'\end{array}}{\Delta \vdash^{t} \texttt{def}\ f\_name(p_1, \ldots, p_n)\ \texttt{do}\ e\ \texttt{end}}\ \text{(T\_DEF\_ANY)}$$

El segundo es que se debe reemplazar el subtipado por el subtipado consistente en T_DEF:

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \emptyset;\emptyset;\tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset;\Gamma_{n-1};\tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n\end{array}}{\Delta \vdash^{t} \texttt{def}\ f\_name(p_1, \ldots, p_n)\ \texttt{do}\ e\ \texttt{end}}\ \text{(T\_DEF)}$$

C.7. Reglas para el chequeo de expresiones

A continuación se presentan las reglas de chequeo de expresiones, enfatizando los cambios respecto al sistema estático.

$$\frac{}{\Delta;\Gamma \vdash^{t} l : type(l) \Rightarrow \emptyset}\ \text{(T\_LIT)} \qquad \frac{\Gamma[x] = \tau}{\Delta;\Gamma \vdash^{t} x : \tau \Rightarrow \emptyset}\ \text{(T\_VAR)}$$

$$\frac{\Delta;\Gamma \vdash^{t} e : \tau \Rightarrow \Gamma' \qquad \widetilde{uniOpType}(\ominus, \tau) = \sigma}{\Delta;\Gamma \vdash^{t} \ominus e : \sigma \Rightarrow \Gamma'}\ \text{(T\_UNI\_OP)}$$

$$\frac{\Delta;\Gamma \vdash^{t} e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \vdash^{t} e_2 : \tau_2 \Rightarrow \Gamma_2 \qquad \widetilde{binOpType}(\oplus, \tau_1, \tau_2) = \sigma}{\Delta;\Gamma \vdash^{t} e_1 \oplus e_2 : \sigma \Rightarrow \Gamma_1 \dagger \Gamma_2}\ \text{(T\_BIN\_OP)}$$

$$\frac{\Delta;\Gamma \vdash^{t} e : \tau \Rightarrow \Gamma' \qquad \Gamma;\tau \vdash^{pt} e : \sigma \Rightarrow \Gamma''}{\Delta;\Gamma \vdash^{t} p\ =\ e : \sigma \Rightarrow \Gamma' \dagger \Gamma''}\ \text{(T\_MATCH)}$$

$$\frac{}{\Delta;\Gamma \vdash^{t} [\,] : [\,] \Rightarrow \emptyset}\ \text{(T\_ELIST)} \qquad \frac{\Delta;\Gamma \vdash^{t} e_i : \tau_i \Rightarrow \Gamma_i \quad i = 1,2 \qquad [\tau_1] \tilde{\nabla} \tau_2 = \sigma}{\Delta;\Gamma \vdash^{t} [e_1 \,|\, e_2] : \sigma \Rightarrow \Gamma_1 \dagger \Gamma_2}\ \text{(T\_CONS)}$$

$$\frac{\Delta;\Gamma \vdash^{t} e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \cdots \qquad \Delta;\Gamma \vdash^{t} e_n : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^{t} \{e_1, \ldots, e_n\} : \{\tau_1, \ldots, \tau_n\} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n}\ \text{(T\_TUPLE)}$$

$$\frac{\Delta;\Gamma \vdash^{t} e_1 : \tau_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Delta;\Gamma \vdash^{t} e_n : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^{t} \%\{(\overline{k_i \Rightarrow e_i})_{i \in [n]}\} : \%\{(\overline{k_i \Rightarrow \tau_i})_{i \in [n]}\} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n}\ \text{(T\_MAP)}$$

$$\frac{\Delta;\Gamma \vdash^{t} e_1 : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \dagger \Gamma_1 \vdash^{t} e_2 : \tau_2 \Rightarrow \Gamma_2}{\Delta;\Gamma \vdash^{t} e_1 ; e_2 : \tau_2 \Rightarrow \Gamma_1 \dagger \Gamma_2}\ \text{(T\_SEQ)}$$

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \quad \tau \lesssim \texttt{boolean} \quad \Delta;\Gamma \dagger \Gamma' \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \quad \Delta;\Gamma \dagger \Gamma' \vdash^t e_2 : \tau_2 \Rightarrow \Gamma_2 \quad \tau_1 \tilde{\vee} \tau_2 = \sigma}{\Delta;\Gamma \vdash^t \texttt{if}\ e\ \texttt{do}\ e_1\ \texttt{else}\ e_2\ \texttt{end} : \sigma \Rightarrow \Gamma'} \text{ (T\_IFE)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \\ \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_1 \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1' \quad \cdots \quad \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_n \vdash^t e_n : \tau_n \Rightarrow \Gamma_n' \\ \sigma = \widetilde{\bigvee}_{i=1}^{n} \tau_i\end{array}}{\Delta;\Gamma \vdash^t \texttt{if}\ e_1\ \texttt{do}\ e_2\ \texttt{else}\ e_3\ \texttt{end} : \sigma \Rightarrow \Gamma'} \text{ (T\_CASE)}$$

$$\text{(T\_ANON)}\ \frac{\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0)}{\Delta;\Gamma \vdash^t \&f\_name/n : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \emptyset} \qquad \frac{(f\_name, n) \notin \text{dom}(\Delta)}{\Delta;\Gamma \vdash^t \&f\_name/n : (\text{any}_1, .., \text{any}_n) \to \text{any} \Rightarrow \emptyset} \text{ (T\_ANON\_ANY)}$$

$$\text{(T\_DCALL)}\ \frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \Delta;\Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots\ \Delta;\Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n \\ \sigma_1 \lesssim \tau_1 \ \cdots\ \sigma_n \lesssim \tau_n\end{array}}{\Delta;\Gamma \vdash^t f\_name(e_1, \ldots, e_n) : \tau_0 \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \qquad \frac{\begin{array}{c}\Delta;\Gamma \vdash^t e : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \Gamma_0 \\ \Delta;\Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots\ \Delta;\Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n \\ \sigma_1 \lesssim \tau_1 \ \cdots\ \sigma_n \lesssim \tau_n\end{array}}{\Delta;\Gamma \vdash^t e.(e_1, \ldots, e_n) : \tau_0 \Rightarrow \Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (T\_ACALL)}$$

$$\text{(T\_DCALL\_ANY)}\ \frac{(f\_name, n) \notin \text{dom}(\Delta) \quad \Delta;\Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \ \cdots\ \Delta;\Gamma \vdash^t e_n : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^t f\_name(e_1, ..., e_n) : \text{any} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \qquad \frac{\Delta;\Gamma \vdash^t e : \text{any} \Rightarrow \Gamma_0 \quad \Delta;\Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots\ \Delta;\Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^t a.(e_1, \ldots, e_n) : \text{any} \Rightarrow \Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (T\_ACALL\_ANY)}$$

C.8. Reglas del chequeo-refinamiento de patrones

C.8.1. Reglas de colección de variables

$$\frac{\mathrm{ty}(l) \lesssim \tau}{\Sigma;\Gamma;\tau \vdash^{pc} l \Rightarrow \Gamma}\ \text{(TPC\_LIT)} \qquad \frac{\Sigma[x] = \sigma \quad \tau\ \tilde{\wedge}\ \sigma = \mu}{\Sigma;\Gamma;\tau \vdash^{pc} \hat{}x \Rightarrow \Gamma}\ \text{(TPC\_PIN)} \qquad \frac{}{\Sigma;\Gamma;\tau \vdash^{pc} \_ \Rightarrow \Gamma}\ \text{(TPC\_WILD)}$$

$$\frac{x \notin \Gamma}{\Sigma;\Gamma;\tau \vdash^{pc} x \Rightarrow \Gamma[x \mapsto \tau]}\ \text{(TPC\_VARE)} \qquad \frac{\Gamma[x] = \sigma \quad \tau\ \tilde{\wedge}\ \sigma = \mu}{\Sigma;\Gamma;\tau \vdash^{pc} x \Rightarrow \Gamma[x \mapsto \mu]}\ \text{(TPC\_VARN)}$$

$$\frac{\Sigma;\Gamma;\tau \vdash^{pc} p_1 \Rightarrow \Gamma' \quad \Sigma;\Gamma';[\tau] \vdash^{pc} p_2 \Rightarrow \Gamma''}{\Sigma;\Gamma;[\tau] \vdash^{pc} [p_1 \mid p_2] \Rightarrow \Gamma''}\ \text{(TPC\_CONS)} \qquad \frac{[\,] \lesssim \tau}{\Sigma;\Gamma;\tau \vdash^{pc} [\,] \Rightarrow \Gamma}\ \text{(TPC\_ELIST)}$$

$$\text{(TPC\_TUPLE)}\ \frac{\Sigma;\Gamma;\tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Sigma;\Gamma_{n-1};\tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n}{\Sigma;\Gamma;\{\tau_1,\ldots,\tau_n\} \vdash^{pc} \{p_1,\ldots,p_n\} \Rightarrow \Gamma_n} \qquad \frac{(k_{i_1},..,k_{i_n}) = (l_1,..,l_n)\ \mathrm{con}\ \{i_1,..,i_n\} \subseteq I \quad \Sigma;\Gamma;\tau_{k_{i_1}} \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Sigma;\Gamma_{n-1};\tau_{k_{i_n}} \vdash^{pc} p_n \Rightarrow \Gamma_n}{\Sigma;\Gamma;\ \%\{(\overline{k_i\ \texttt{=>}\ \tau_i})_{i\in I}\} \vdash^{pc} \%\{(\overline{l_i\ \texttt{=>}\ p_i})_{i\in\{1,\ldots,n\}}\} \Rightarrow \Gamma_n}\ \text{(TPC\_MAP)}$$

$$\frac{ground_P(p) = \mu \qquad \Sigma;\Gamma;\mu \vdash^{pc} p \Rightarrow \Gamma_1}{\Sigma;\Gamma_1;\mathsf{any} \vdash^{pc} p \Rightarrow \Gamma_1}\ \text{(TPC\_GROUND)}$$

estando GroundP definido como:

procedure GroundP(p)
  switch p **do**
    case l : **return** Type(l)
    case `[ ]` : **return** `[ ]`
    case `[`p`]` : **return** `[any]`
    case `{`p_1`,...,`p_n`}` : **return** `{any,...,any}`
    case `%{`k_1 `=>` p_1`,...,`k_n `=>` p_n`}` : **return** `%{`k_1 `=>` `any,...,`k_n `=>` `any}`
    default : **fail**
end procedure

C.8.2. Reglas para el juicio de refinamiento en patrones

Los únicos cambios son el reemplazo del supremo por el supremo consistente en TPV_CONS y la nueva regla TPV_GROUND:

$$\text{(TPV\_CONS)}\ \frac{\Sigma;\Gamma;\tau \vdash^{ptv} p : \sigma_1 \quad \Sigma;\Gamma;[\tau] \vdash^{ptv} p_2 : \sigma_2 \quad [\sigma_1]\ \tilde{\nabla}\sigma_2 = \sigma}{\Sigma;\Gamma;[\tau] \vdash^{ptv} [p_1 \mid p_2] : \sigma} \qquad \frac{ground_P(p) = \tau \qquad \Sigma;\Gamma;\tau \vdash^{ptv} p : \sigma}{\Sigma;\Gamma;\mathsf{any} \vdash^{ptv} p : \sigma}\ \text{(TPV\_GROUND)}$$

C.9. Detalles de la prueba de la SGG

La verificación de la SGG en las reglas del sistema gradual es inductiva en la estructura de la derivación y conjunta para todos los lemas. Esto quiere decir que para cualquiera de éstas reglas, es posible asumir la validez de todos los lemas para los juicios de las hipótesis.

Verificación de SGG (DEF) en T_DEF

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \emptyset; \emptyset; \tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset; \Gamma_{n-1}; \tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta; \Gamma_n \vdash^{t} e : \sigma \Rightarrow \Gamma' \quad \sigma \lesssim \tau_0\end{array}}{\Delta \vdash^{t} \texttt{def}\ \ f\_name(p_1, \ldots, p_n)\ \ \texttt{do}\ \ e\ \ \texttt{end}}\ \text{(T\_DEF)}$$

Se parte de la hipótesis $\Delta \lll \Delta'$.

1. **Caso** $(f\_name, n) \notin \mathbf{dom}(\Delta')$: Se aplica inducción respecto a i para deducir $\Gamma_i \lll \Gamma'_i \quad \forall i \in \{0, ..., n\}$ (con la convención de que $\Gamma_0 = \emptyset$):

   a) $i = 0$: De las hipótesis $\emptyset; \emptyset; \mathsf{any} \vdash^{pc} p_1 \Rightarrow \Gamma_1$ y $\tau_1 \lll \mathsf{any}$ se deduce $\Gamma_1 \lll \Gamma'_1$ por SGG (PAT1).

   b) $i = k + 1$: De las hipótesis $\emptyset; \Gamma_k; \mathsf{any} \vdash^{pc} p_{k+1} \Rightarrow \Gamma_{k+1}$, $\tau_{k+1} \lll \mathsf{any}$ y $\Gamma_k \lll \Gamma'_k$ se deduce $\emptyset; \Gamma_k; \mathsf{any} \vdash^{pc} p_{k+1} \Rightarrow \Gamma_{k+1}$ y $\Gamma_{k+1} \lll \Gamma'_{k+1}$ por SGG (PAT1).

   De T_DEF_ANY se deduce $\Delta' \vdash^{t} \texttt{def}\ \ f\_name(p_1, \ldots, p_n)\ \ \texttt{do}\ \ e\ \ \texttt{end}$.

2. **Caso** $\Delta'[(f\_name, n)] = ((\tau'_1, ..., \tau'_n), \tau'_0)$: Por definición de $\lll$ en contextos, se tiene $\tau_i \lll \tau'_i \quad \forall i \in \{0, ..., n\}$. Luego, se aplica inducción respecto a i para deducir $\emptyset; \Gamma_i; \tau_{i+1} \vdash^{pc} p_i \Rightarrow \Gamma_{i+1}$ y $\Gamma_i \lll \Gamma'_i \quad \forall i \in \{0, ..., n\}$ (con la convención de que $\Gamma_0 = \emptyset$):

   a) $i = 0$: De las hipótesis $\emptyset; \emptyset; \tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1, \tau_1 \lll \tau'_1$ se deduce $\Gamma_1 \lll \Gamma'_1$ por SGG (PAT1).

   b) $i = k + 1$: De las hipótesis $\emptyset; \Gamma_k; \tau_{k+1} \vdash^{pc} p_{k+1} \Rightarrow \Gamma_{k+1}$, $\tau_{k+1} \lll \tau'_{k+1}$ y $\Gamma_k \lll \Gamma'_k$ se deduce $\emptyset; \Gamma_k; \tau_{k+1} \vdash^{pc} p_{k+1} \Rightarrow \Gamma_{k+1}$ y $\Gamma_{k+1} \lll \Gamma'_{k+1}$ por SGG (PAT1).

   Aplicando SGG (EXPR) para $\Delta; \Gamma_n \vdash^{t} e : \sigma \Rightarrow \Gamma'$, $\Delta \lll \Delta'$ y $\Gamma \lll \Gamma'$, se deduce $\sigma \lll \sigma'$. Además, de la monotonía de $\lesssim$ junto con $(\sigma, \tau_0) \in \lesssim$ y $((\sigma, \tau_0), (\sigma', \tau'_0)) \in \lll^2$ se concluye $(\sigma', \tau'_0) \in \lesssim$, es decir, $\sigma' \lesssim \tau'_0$.

   Finalmente, por T_DEF se concluye $\Delta' \vdash^{t} \texttt{def}\ \ f\_name(p_1, \ldots, p_n)\ \ \texttt{do}\ \ e\ \ \texttt{end}$.

Verificación de SGG (DEF) en T_DEF_ANY

$$\frac{\begin{array}{c}(f\_name, n) \notin \mathrm{dom}(\Delta) \\ \emptyset; \emptyset; \mathsf{any} \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset; \Gamma_{n-1}; \mathsf{any} \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta; \Gamma_n \vdash^{t} e : \sigma \Rightarrow \Gamma'\end{array}}{\Delta \vdash^{t} \texttt{def}\ \ f\_name(p_1, \ldots, p_n)\ \ \texttt{do}\ \ e\ \ \texttt{end}}\ \text{(T\_DEF\_ANY)}$$

Se parte de la hipótesis $\Delta \bar{\lll} \Delta'$.
Por definición de $\bar{\lll}$, $(f\_name, n) \notin \text{dom}(\Delta)$ implica $(f\_name, n) \notin \text{dom}(\Delta')$. Por lo tanto, aplica T_DEF_ANY con las restantes hipótesis idénticas, para deducir $\Delta' \vdash^t \texttt{def}\ f\_name(p_1, ..., p_n)\ \texttt{do}\ e\ \texttt{end}$.

Verificación de SGG (EXPR) en T_ACALL_ANY

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e : \texttt{any} \Rightarrow \Gamma_0 \\ \Delta;\Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots\ \Delta;\Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n\end{array}}{\Delta;\Gamma \vdash^t e.(e_1, ..., e_n) : \texttt{any} \Rightarrow \Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n}\ \text{(T\_ACALL\_ANY)}$$

Se parte de las hipótesis $\Delta \bar{\lll} \Delta'$ y $\Gamma \lll \Gamma'$.
Aplicando recursivamente el lema para e se deduce que $\Delta';\Gamma' \vdash^t e' : \sigma \Rightarrow \Gamma'_0$ con $\texttt{any} \lll \sigma$ y $\Gamma_0 \lll \Gamma'_0$. En particular, $\sigma = \texttt{any}$. Además, para cada $i \in \{1, ..., n\}$ se aplica inducción para deducir $\Delta;\Gamma \vdash^t e_i : \sigma'_i \Rightarrow \Gamma'_i$, $\sigma_i \lll \sigma'_i$ y $\Gamma_i \lll \Gamma'_i$ a partir de $\Delta;\Gamma \vdash^t e_i : \sigma_i \Rightarrow \Gamma_i$.
De T_ACALL_ANY se deduce $\Delta';\Gamma' \vdash^t e.(e_1, ..., e_n) : \texttt{any} \Rightarrow \Gamma'_0 \dagger \Gamma'_1 \dagger \cdots \dagger \Gamma'_n$. Además, como $\dagger$ es un operador monótono se tiene $\Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n \lll \Gamma'_0 \dagger \Gamma'_1 \dagger \cdots \dagger \Gamma'_n$.

Verificación de SGG (EXPR) en T_ANON

$$\frac{\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0)}{\Delta;\Gamma \vdash^t \&f\_name/n : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \emptyset}\ \text{(T\_ANON)}$$

Se parte de la hipótesis $\Delta \bar{\lll} \Delta'$.

1. **Caso** $(f\_name, n) \notin \textbf{dom}(\Delta')$: Por T_DEF_ANY se deduce $\Delta;\Gamma \vdash^t \&f\_name/n : (\texttt{any}_1, .., \texttt{any}_n) \to \texttt{any} \Rightarrow \emptyset$ y por definición $(\tau_1, .., \tau_n) \to \tau_0 \lll (\texttt{any}_1, .., \texttt{any}_n) \to \texttt{any}$.

2. **Caso** $\Delta'[(f\_name, n)] = ((\tau'_1, ..., \tau'_n), \tau'_0)$: Por, por T_DEF se concluye $\Delta;\Gamma \vdash^t \&f\_name/n : (\tau'_1, .., \tau'_n) \to \tau'_0 \Rightarrow \emptyset$ y además $(\tau_1, .., \tau_n) \to \tau_0 \lll (\tau'_1, .., \tau'_n) \to \tau'_0$, ya que $\tau_i \lll \tau'_i \quad \forall i \in \{0, ..., n\}$ por definición de $\Delta[(f\_name, n)] \lll \Delta'[(f\_name, n)]$.

Verificación de SGG (EXPR) en T_ANON_ANY

$$\frac{(f\_name, n) \notin \text{dom}(\Delta)}{\Delta;\Gamma \vdash^t \&f\_name/n : (\texttt{any}_1, .., \texttt{any}_n) \to \texttt{any} \Rightarrow \emptyset}\ \text{(T\_ANON\_ANY)}$$

Análogo al T_DEF_ANY.

Verificación de SGG (EXPR) en T_DCALL

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \Delta;\Gamma \vdash^t e_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots\ \Delta;\Gamma \vdash^t e_n : \sigma_n \Rightarrow \Gamma_n \\ \sigma_1 \lesssim \tau_1 \ \cdots\ \sigma_n \lesssim \tau_n\end{array}}{\Delta;\Gamma \vdash^t f\_name(e_1, ..., e_n) : \tau_0 \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n}\ \text{(T\_DCALL)}$$

Se parte de las hipótesis $\Delta \bar{\lll} \Delta'$ y $\Gamma \lll \Gamma'$.

1. **Caso** $(f\_name, n) \notin \mathbf{dom}(\Delta')$: Para cada $i \in \{1, ..., n\}$ se aplica inducción para deducir $\Delta; \Gamma \vdash^t e_i : \sigma'_i \Rightarrow \Gamma'_i$, $\sigma_i \lll \sigma'_i$ y $\Gamma_i \lll \Gamma'_i$ a partir de $\Delta; \Gamma \vdash^t e_i : \sigma_i \Rightarrow \Gamma_i$.

   De T_DCALL_ANY se deduce $\Delta'; \Gamma' \vdash^t f\_name(e_1, ..., e_n) : \mathsf{any} \Rightarrow \Gamma'_1 \dagger \cdots \dagger \Gamma'_n$. Además, como $\dagger$ es un operador monótono se tiene $\Gamma_1 \dagger \cdots \dagger \Gamma_n \lll \Gamma'_1 \dagger \cdots \dagger \Gamma'_n$.

2. **Caso** $\Delta'[(f\_name, n)] = ((\tau'_1, ..., \tau'_n), \tau'_0)$: Para cada $i \in \{1, ..., n\}$ se aplica inducción para deducir $\Delta; \Gamma \vdash^t e_i : \sigma'_i \Rightarrow \Gamma'_i$, $\sigma_i \lll \sigma'_i$ y $\Gamma_i \lll \Gamma'_i$ a partir de $\Delta; \Gamma \vdash^t e_i : \sigma_i \Rightarrow \Gamma_i$.

   Por monotonía de $\lesssim$ para cada i respecto a $\lll$, se deduce $\sigma_i \lesssim \tau'_i$.

   Finalmente, por T_DCALL_DEF se deduce $\Delta'; \Gamma' \vdash^t f\_name(e_1, ..., e_n) : \tau_0 \Rightarrow \Gamma'_1 \dagger \cdots \dagger \Gamma'_n$. Además, como $\dagger$ es un operador monótono se tiene $\Gamma_1 \dagger \cdots \dagger \Gamma_n \lll \Gamma'_1 \dagger \cdots \dagger \Gamma'_n$.

Verificación de SGG (EXPR) en T_DCALL_ANY

$$\frac{(f\_name, n) \notin \mathrm{dom}(\Delta) \qquad \Delta; \Gamma \vdash^t e_1 : \tau_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta; \Gamma \vdash^t e_n : \tau_n \Rightarrow \Gamma_n}{\Delta; \Gamma \vdash^t f\_name(e_1, ..., e_n) : \mathsf{any} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (T\_DCALL\_ANY)}$$

Se parte de la hipótesis $\Delta \lll \Delta'$ y $\Gamma \lll \Gamma'$.
Por definición de $\lll$, $(f\_name, n) \notin \mathrm{dom}(\Delta)$ implica $(f\_name, n) \notin \mathrm{dom}(\Delta')$. Además, Para cada $i \in \{1, ..., n\}$, $\Delta; \Gamma \vdash^t e_i : \tau_i \Rightarrow \Gamma_i$ implica $\Gamma_i \lll \Gamma'_i$ por SGG (EXPR). Luego, aplica T_DCALL _ANY y se tiene $\Delta'; \Gamma' \vdash^t f\_name(e_1, ..., e_n) : \mathsf{any} \Rightarrow \Gamma'_1 \dagger \cdots \dagger \Gamma'_n$. Además, como $\dagger$ es un operador monótono se tiene $\Gamma_1 \dagger \cdots \dagger \Gamma_n \lll \Gamma'_1 \dagger \cdots \dagger \Gamma'_n$.

Verificación de SGG (PAT1) en TPC_GROUND

$$\frac{ground_P(p) = \mu \qquad \Sigma; \Gamma; \mu \vdash^{pc} p \Rightarrow \Gamma_1}{\Sigma; \Gamma_1; \mathsf{any} \vdash^{pc} p \Rightarrow \Gamma_1} \text{ (TPC\_GROUND)}$$

Se parte de las hipótesis $\Sigma \lll \Sigma'$, $\Gamma \lll \Gamma'$ y $\mathsf{any} \lll \tau'$. De ésta última se deduce $\tau' = \mathsf{any}$.

Por SGG (PAT1) en $\Sigma; \Gamma; \mu \vdash^{pc} p \Rightarrow \Gamma_1$, se deduce $\Sigma'; \Gamma'; \mu \vdash^{pc} p \Rightarrow \Gamma'_1$ y $\Gamma_1 \lll \Gamma'_1$. Luego, por TPC_GROUND se deduce $\Sigma'; \Gamma'_1; \mathsf{any} \vdash^{pc} p \Rightarrow \Gamma'_1$.

Verificación de SGG (PAT2) en TPV_GROUND

$$\frac{ground_P(p) = \tau \qquad \Sigma; \Gamma; \tau \vdash^{ptv} p : \sigma}{\Sigma; \Gamma; \mathsf{any} \vdash^{ptv} p : \sigma} \text{ (TPV\_GROUND)}$$

Análogo al anterior.

Verificación de SGG (PAT1) en TPC_CONS, TPC_TUPLE, TPC_MAP

Se prueba para TPC_CONS; en TPC_TUPLE y TPC_MAP la prueba es completamente análoga.

$$\frac{\Sigma; \Gamma; \tau \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \Sigma; \Gamma_1; [\tau] \vdash^{pc} p_2 \Rightarrow \Gamma_2}{\Sigma; \Gamma; [\tau] \vdash^{pc} [p_1 \,|\, p_2] \Rightarrow \Gamma_2} \text{ (TPC\_CONS)}$$

Se parte de las hipótesis $\Sigma \lll \Sigma'$ y $\Gamma \lll \Gamma'$ y $[\tau] \lll \tau'$. Se abren dos casos según τ':

- $[\tau']$ con $\tau \lll \tau'$: Es el caso no interesante, y se prueba por aplicación inductiva de SGG (PAT1) en una secuencia de dos casos:

  1. De $\Sigma; \Gamma; \tau \vdash^{pc} p_1 \Rightarrow \Gamma_1$ se deduce $\Sigma'; \Gamma'; \tau' \vdash^{pc} p_1 \Rightarrow \Gamma'_1$ con $\Gamma_1 \lll \Gamma'_1$.
  2. Gracias a 1., de $\Sigma; \Gamma_1; [\tau] \vdash^{pc} p_2 \Rightarrow \Gamma_2$ se deduce $\Sigma'; \Gamma'_1; [\tau'] \vdash^{pc} p_2 \Rightarrow \Gamma'_2$ y $\Gamma_2 \lll \Gamma'_2$.

  Finalmente por aplicación de TPC_CONS a 1. y 2. se deduce $\Sigma'; \Gamma'; [\tau'] \vdash^{pc} [p_1 \mid p_2] \Rightarrow \Gamma'_2$ donde $\Gamma_2 \lll \Gamma'_2$.

- any: Es claro que el argumento anterior aplica a [any] ya que $\tau \lll$ any, para concluir $\Sigma'; \Gamma'; [\text{any}] \vdash^{pc} [p_1 \mid p_2] \Rightarrow \Gamma'_1$.

Finalmente, por aplicación de TPC_GROUND:

$$\frac{ground_P([p_1 \mid p_2]) = [\text{any}] \qquad \Sigma'; \Gamma'; [\text{any}] \vdash^{pc} [p_1 \mid p_2] \Rightarrow \Gamma'_1}{\Sigma'; \Gamma'; \text{any} \vdash^{pc} [p_1 \mid p_2] \Rightarrow \Gamma'_1} \text{(TPC\_GROUND)}$$

lo que valida el juicio $\Sigma'; \Gamma'; \text{any} \vdash^{pc} [p_1 \mid p_2] \Rightarrow \Gamma'_1$.

Verificación de SGG (PAT2) en TPV_CONS, TPV_TUPLE, TPV_MAP

Se prueba para TPV_CONS; en TPV_TUPLE y TPV_MAP la prueba es muy similar.

$$\frac{\Sigma; \Gamma; \tau \vdash^{ptv} p : \sigma_1 \quad \Sigma; \Gamma; [\tau] \vdash^{ptv} p_2 : \sigma_2 \quad [\sigma_1] \tilde{\nabla} \sigma_2 = \sigma}{\Sigma; \Gamma; [\tau] \vdash^{ptv} [p_1 \mid p_2] : \sigma} \text{(TPV\_CONS)}$$

Se parte de las hipótesis $\Sigma \lll \Sigma'$ y $\Gamma \lll \Gamma'$ y $[\tau] \lll \tau'$. Se abren dos casos según τ':

- $[\tau']$ con $\tau \lll \tau'$: Es el caso no interesante, y se prueba por aplicación inductiva de SGG (PAT1) en dos casos independientes y la monotonía de $\tilde{\nabla}$:

  1. De $\Sigma; \Gamma; \tau \vdash^{ptv} p_1 : \sigma_1$ se deduce $\Sigma'; \Gamma'; \tau' \vdash^{ptv} p_1 : \sigma'_1$ con $\sigma_1 \lll \sigma'_1$.
  2. Gracias a 1., de $\Sigma; \Gamma; [\tau] \vdash^{ptv} p_2 : \sigma_2$ se deduce $\Sigma'; \Gamma'; [\tau'] \vdash^{ptv} p_2 : \sigma'_2$ y $\sigma_2 \lll \sigma'_2$.
  3. Por la monotonía de $\tilde{\nabla}$ respecto a $\lll$, se tiene $[\sigma_1] \tilde{\nabla} \sigma_2 \lll [\sigma'_1] \tilde{\nabla} \sigma'_2$ (y en particular $[\sigma'_1] \tilde{\nabla} \sigma'_2$ está definido).

  Finalmente por aplicación de TPV_CONS a 1., 2.y 3. se deduce la validez de $\Sigma'; \Gamma'; [\tau'] \vdash^{ptv} [p_1 \mid p_2] : [\sigma'_1] \tilde{\nabla} \sigma'_2 = \sigma'$.

- any: Es claro que el argumento anterior aplica a [any] ya que $\tau \lll$ any, para concluir $\Sigma'; \Gamma'; [\text{any}] \vdash^{ptv} [p_1 \mid p_2] : \sigma'$. Finalmente, por aplicación de TPV_GROUND:

$$\frac{ground_P([p_1 \mid p_2]) = [\text{any}] \qquad \Sigma'; \Gamma'; [\text{any}] \vdash^{ptv} [p_1 \mid p_2] : \sigma'}{\Sigma'; \Gamma'; \text{any} \vdash^{ptv} [p_1 \mid p_2] : \sigma'} \text{(TPV\_GROUND)}$$

lo que valida el juicio $\Sigma'; \Gamma'; \text{any} \vdash^{ptv} [p_1 \mid p_2] : \sigma'$.

Apéndice D

Sistema de traducción

D.1. Reglas para la traducción de un programa

$$\frac{\Delta \vdash^t d_1 \rightsquigarrow d_1' \quad \ldots \quad \Delta \vdash^t d_n \rightsquigarrow d_n'}{\begin{array}{l}\Delta \vdash^t \texttt{defmodule Program do } d_1\texttt{, } \ldots\texttt{, } d_n \texttt{ end} \\ \rightsquigarrow \texttt{defmodule Program do } d_1'\texttt{, } \ldots\texttt{, } d_n' \texttt{ end}\end{array}} \text{ (TRANSLATE\_PROG\_AUX)}$$

$$\frac{\vdash^c m \Rightarrow \Delta \qquad \Delta \vdash^{ts} m \rightsquigarrow m'}{\vdash^t m \rightsquigarrow m'} \text{ (TRANSLATE\_PROG)}$$

donde las reglas para la traducción de declaraciones son:

$$\frac{}{\Delta \vdash s \rightsquigarrow s} \text{ (TRANSLATE\_SPEC)}$$

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \\ \emptyset;\emptyset;\tau_1 \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset;\Gamma_{n-1};\tau_n \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta;\Gamma_n \vdash^t e : \sigma \Rightarrow \Gamma' \quad \sigma \lesssim \tau_0\end{array}}{\begin{array}{l}\Delta \vdash^t \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end} \\ \rightsquigarrow \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } (e \mid \sigma \sim\!\!\gg \tau_0) \texttt{ end}\end{array}} \text{ (TRANSLATE\_DEF)}$$

$$\frac{\begin{array}{c}(f\_name, n) \notin \text{dom}(\Delta) \\ \emptyset;\emptyset;\text{any} \vdash^{pc} p_1 \Rightarrow \Gamma_1, \quad \ldots, \quad \emptyset;\Gamma_{n-1};\text{any} \vdash^{pc} p_n \Rightarrow \Gamma_n\end{array}}{\begin{array}{l}\Delta \vdash^t \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end} \\ \rightsquigarrow \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } (e \mid \sigma \sim\!\!\gg \text{any}) \texttt{ end}\end{array}} \text{ (TRANSLATE\_DEF\_ANY)}$$

D.2. Reglas para la traducción de expresiones

$$\frac{}{\Delta;\Gamma \vdash^t l \rightsquigarrow l : type(l) \Rightarrow \emptyset} \text{ (TRANSLATE\_LIT)} \qquad \frac{\Gamma[x] = \tau}{\Delta;\Gamma \vdash^t x \rightsquigarrow x : \tau \Rightarrow \emptyset} \text{ (TRANSLATE\_VAR)}$$

$$\frac{\Delta;\Gamma \vdash^t e : \tau \Rightarrow \Gamma' \qquad \widetilde{uniOpType}(\ominus, \tau) = \sigma}{\Delta;\Gamma \vdash^t e \rightsquigarrow \ominus(e \mid \tau \sim\!\!\gg \tau^{\ominus}) : \sigma \Rightarrow \Gamma'} \text{ (TRANSLATE\_UNI\_OP)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \vdash^t e_2 \rightsquigarrow e_2' : \tau_2 \Rightarrow \Gamma_2 \\ \tau_1 \leftharpoonup \tau_1^{\oplus} = \tau_1' \quad \tau_2 \leftharpoonup \tau_2^{\oplus} = \tau_2' \\ \widetilde{binOpType}(\oplus, \tau_1, \tau_2) = \sigma \quad binOpType(\oplus, \tau_1', \tau_2') = \sigma'\end{array}}{\Delta;\Gamma \vdash^t e_1 \oplus e_2 \rightsquigarrow ((e_1' \mid \tau_1 \sim\!\!\gg \tau_1') \oplus (e_2' \mid \tau_2 \sim\!\!\gg \tau_2') \mid \sigma' \sim\!\!\gg \sigma) : \sigma \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (TRANSLATE\_BIN\_OP)}$$

$$\frac{\Delta;\Gamma \vdash^t e \rightsquigarrow e' : \tau \Rightarrow \Gamma' \qquad \Gamma;\tau \vdash^{pt} e : \sigma \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^t p \;=\; e \rightsquigarrow p \;=\; e' : \sigma \Rightarrow \Gamma' \dagger \Gamma''} \text{ (TRANSLATE\_MATCH)}$$

$$\frac{}{\Delta;\Gamma \vdash^t [\,] \rightsquigarrow [\,] : [\,] \Rightarrow \emptyset} \text{ (TRANSLATE\_ELIST)}$$

$$\frac{\Delta;\Gamma \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \vdash^t e_2 \rightsquigarrow e_2' : \tau_2 \Rightarrow \Gamma_2 \qquad [\tau_1]\widetilde{\vee}\tau_2 = \sigma}{\Delta;\Gamma \vdash^t [e_1 | e_2] \rightsquigarrow [(e_1' \mid \tau_1 \sim\!\!\gg \sigma) | (e_2' \mid \tau_2 \sim\!\!\gg [\sigma])] : [\sigma] \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (TRANSLATE\_CONS)}$$

$$\frac{\Delta;\Gamma \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1 \qquad \cdots \qquad \Delta;\Gamma \vdash^t e_n \rightsquigarrow e_n' : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^t \{e_1, \ldots, e_n\} \rightsquigarrow \{e_1', \ldots, e_n'\} : \{\tau_1, \ldots, \tau_n\} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (TRANSLATE\_TUPLE)}$$

$$\frac{\Delta;\Gamma \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Delta;\Gamma \vdash^t e_n \rightsquigarrow e_n' : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^t \%\{(\overline{k_i \;=>\; e_i})_{i\in[n]}\} \rightsquigarrow \%\{(\overline{k_i \;=>\; e_i'})_{i\in[n]}\} : \%\{(\overline{k_i \;=>\; \tau_i})_{i\in[n]}\} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (TRANSLATE\_MAP)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e \rightsquigarrow e' : \tau \Rightarrow \Gamma' \quad \tau \lesssim \texttt{boolean} \\ \Delta;\Gamma \dagger \Gamma' \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1 \quad \Delta;\Gamma \dagger \Gamma' \vdash^t e_2 \rightsquigarrow e_2' : \tau_2 \Rightarrow \Gamma_2 \quad \tau_1\widetilde{\vee}\tau_2 = \sigma\end{array}}{\begin{array}{l}\Delta;\Gamma \vdash^t \texttt{if } e \texttt{ do } e_1 \texttt{ else } e_2 \texttt{ end} \\ \rightsquigarrow \texttt{if } (e' \mid \tau \sim\!\!\gg \texttt{boolean}) \texttt{ do } (e_1' \mid \tau_1 \sim\!\!\gg \sigma) \texttt{ else } (e_2' \mid \tau_2 \sim\!\!\gg \sigma) \texttt{ end} : \sigma \Rightarrow \Gamma'\end{array}} \text{ (TRANSLATE\_IFE)}$$

$$\frac{\Delta;\Gamma \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \dagger \Gamma_1 \vdash^t e_2 \rightsquigarrow e_2' : \tau_2 \Rightarrow \Gamma_2}{\Delta;\Gamma \vdash^t e_1 ; e_2 \rightsquigarrow e_1' ; e_2' : \tau_2 \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (TRANSLATE\_SEQ)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e \rightsquigarrow e' : \tau \Rightarrow \Gamma' \\ \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Gamma \dagger \Gamma';\emptyset;\tau \vdash^{pc} p_n \Rightarrow \Gamma_n \\ \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_1 \vdash^t e_1 \rightsquigarrow e_1' : \tau_1 \Rightarrow \Gamma_1' \quad \cdots \quad \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_n \vdash^t e_n \rightsquigarrow e_n' : \tau_n \Rightarrow \Gamma_n' \\ \sigma = \widetilde{\bigvee}_{i=1}^{n} \tau_i\end{array}}{\begin{array}{l}\Delta;\Gamma \vdash^t \texttt{case } e \texttt{ do } p_1 \texttt{ -> } e_1\texttt{; } \ldots\texttt{; } p_n \texttt{ -> } e_n \texttt{ end} \\ \rightsquigarrow \texttt{case } e' \texttt{ do } p_1 \texttt{ -> } (e_1' \mid \tau_1 \sim\!\!\gg \sigma)\texttt{; } \ldots\texttt{; } p_n \texttt{ -> } (e_n' \mid \tau_n \sim\!\!\gg \sigma) \texttt{ end} : \sigma \Rightarrow \Gamma'\end{array}} \text{ (TRANSLATE\_CASE)}$$

$$\frac{\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0)}{\Delta;\Gamma \vdash^t \&f\_name/n \rightsquigarrow \&f\_name/n : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \emptyset} \text{ (TRANSLATE\_ANON)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^t e \rightsquigarrow e' : (\tau_1, .., \tau_n) \to \tau_0 \Rightarrow \Gamma_0 \\ \Delta;\Gamma \vdash^t e_1 \rightsquigarrow e_1' : \sigma_1 \Rightarrow \Gamma_1 \;\cdots\; \Delta;\Gamma \vdash^t e_n \rightsquigarrow e_n' : \sigma_n \Rightarrow \Gamma_n \\ \sigma_1 \lesssim \tau_1 \;\cdots\; \sigma_n \lesssim \tau_n\end{array}}{\Delta;\Gamma \vdash^t e.(e_1, \ldots, e_n) \rightsquigarrow e'.((e_1' \mid \sigma_1 \sim\!\!\gg \tau_1), \ldots, (e_n' \mid \sigma_n \sim\!\!\gg \tau_n)) : \tau_0 \Rightarrow \Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (TRANSLATE\_ACALL)}$$

$$\frac{\Delta;\Gamma \vdash^t e \rightsquigarrow e' : \mathsf{any} \Rightarrow \Gamma_0 \quad \Delta;\Gamma \vdash^t e_1 \rightsquigarrow e'_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^t e_n \rightsquigarrow e'_n : \sigma_n \Rightarrow \Gamma_n \quad e'' = (e' \mid \mathsf{any} \sim\!> (\mathsf{any}, .., \mathsf{any}_n) \to \mathsf{any})}{\Delta;\Gamma \vdash^t e.(e_1, \ldots, e_1) \rightsquigarrow e''.((e'_1 \mid \sigma_1 \sim\!\!\gg \mathsf{any}), \ldots, (e'_n \mid \sigma_n \sim\!\!\gg \mathsf{any})) : \mathsf{any} \Rightarrow \Gamma_0 \dagger \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (TRANSLATE\_ACALL\_ANY)}$$

$$\frac{\Delta[(f\_name, n)] = ((\tau_1, \ldots, \tau_n), \tau_0) \quad \Delta;\Gamma \vdash^t e_1 \rightsquigarrow e'_1 : \sigma_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^t e_n \rightsquigarrow e'_n : \sigma_n \Rightarrow \Gamma_n \quad \sigma_1 \lesssim \tau_1 \ \cdots \ \sigma_n \lesssim \tau_n}{\Delta;\Gamma \vdash^t f\_name(e_1, \ldots, e_n) \rightsquigarrow f\_name((e_1 \mid \sigma_1 \sim\!\!\gg \tau_1), \ldots, (e_n \mid \sigma_n \sim\!\!\gg \tau_n)) : \tau_0 \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (TRANSLATE\_DCALL)}$$

$$\frac{(f\_name, n) \notin \mathrm{dom}(\Delta) \quad \Delta;\Gamma \vdash^t e_1 \rightsquigarrow e'_1 : \tau_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^t e_n \rightsquigarrow e'_n : \tau_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^t f\_name(e_1, \ldots, e_n) \rightsquigarrow f\_name((e_1 \mid \tau_1 \sim\!\!\gg \mathsf{any}), \ldots, (e_n \mid \tau_n \sim\!\!\gg \mathsf{any})) : \mathsf{any} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (T\_DCALL\_ANY)}$$

Apéndice E

Semántica de evaluación

E.1. Reglas de evaluación de un programa

Las regla que definen la evaluación de un programa son:

$$\frac{\vdash^* m \Rightarrow \Delta \qquad \Delta[(\texttt{main}, 0)] = e \qquad \Delta; \emptyset \vdash^* e \Downarrow r \Rightarrow \Gamma}{\vdash^* m \Downarrow r} \text{ (E\_PROG)}$$

donde además se deben definir las reglas para la colección dinámica

$$\frac{\emptyset \vdash^* d_1 \Rightarrow \Delta_1 \quad \ldots \quad \Delta_{n-1} \vdash^* d_n \Rightarrow \Delta_n}{\vdash^* \texttt{defmodule Program do } d_1\texttt{, } \ldots\texttt{, } d_n \texttt{ end} \Rightarrow \Delta_n} \text{ (ECOLLECT\_PROG)}$$

$$\frac{}{\Delta \vdash^* \texttt{@spec } f\_name(\tau_1, \ldots, \tau_n) \texttt{ :: } \tau_0 \Rightarrow \Delta} \text{ (EECOLLECT\_SPEC)}$$

$$\frac{(f\_name, n) \notin \Delta}{\Delta \vdash^* \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end} \Rightarrow \Delta[(f\_name, n) \mapsto (((p_1, \ldots, p_n), e))]} \text{ (ECOLLECT\_EDEF)}$$

$$\frac{\Delta[(f\_name, n)] = ((\vec{p}^1, e^1) \cdots, (\vec{p}^k, e^k))}{\Delta \vdash^* \texttt{def } f\_name(p_1, \ldots, p_n) \texttt{ do } e \texttt{ end} \Rightarrow \Delta[(f\_name, n) \mapsto ((\vec{p}^1, e^1) \cdots, (\vec{p}^k, e^k), ((p_1, \ldots, p_n), e))]} \text{ (ECOLLECT\_DEF)}$$

E.2. Reglas de evaluación de expresiones

Primero se exhiben todos los casos cuyas hipótesis de evaluación son todas exitosas. En caso de que haya alguna hipótesis que retorne error, aplican las reglas de propagación que se describen después.

$$\frac{}{\Delta; \Gamma \vdash^* l \Downarrow l \Rightarrow \emptyset} \text{ (E\_LIT)} \qquad \frac{\Gamma[x] = v}{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \emptyset} \text{ (E\_VAR)}$$

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \qquad \mathit{uniOpVal}(\ominus, v) = v'}{\Delta; \Gamma \vdash^* \ominus e \Downarrow v' \Rightarrow \Gamma'} \text{ (E\_UNI\_OP)}$$

$$\frac{\Delta; \Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \qquad \Delta; \Gamma \vdash^* e_2 \Downarrow v_2 \Rightarrow \Gamma_2 \qquad \mathit{binOpVal}(\oplus, v_1, v_2) = v_3}{\Delta; \Gamma \vdash^* e_1 \oplus e_2 \Downarrow v_3 \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (E\_BIN\_OP)}$$

$$\frac{\Delta; \Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \qquad \Delta; \Gamma \vdash^* e_2 \Downarrow v_2 \Rightarrow \Gamma_2 \qquad \mathit{binOpVal}(\oplus, v_1, v_2) = v_3}{\Delta; \Gamma \vdash^* e_1 \oplus e_2 \Downarrow v_3 \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (E\_BIN\_OP)}$$

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \qquad \mathit{type}(v) \not\leq \texttt{boolean}}{\Delta; \Gamma \vdash^* \texttt{not } e \Downarrow \texttt{ArgumentError} \Rightarrow \emptyset} \text{ (E\_UNI\_OP\_ARGUMENT\_ERROR1)}$$

$$\frac{\Delta; \Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \quad \Delta; \Gamma \vdash^* e_2 \Downarrow v_2 \Rightarrow \Gamma_2 \quad (v_1, v_2) \not\leq (\texttt{string}, \texttt{string})}{\Delta; \Gamma \vdash^* e_1 <> e_2 \Downarrow \texttt{ArgumentError} \Rightarrow \emptyset} \text{ (E\_BIN\_OP\_ARGUMENT\_ERROR2)}$$

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \qquad \mathit{type}(v) \not\leq \texttt{number}}{\Delta; \Gamma \vdash^* \texttt{-}e \Downarrow \texttt{ArgumentError} \Rightarrow \emptyset} \text{ (E\_UNI\_OP\_ARITHMETIC\_ERROR1)}$$

$$\frac{\Delta; \Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \quad \Delta; \Gamma \vdash^* e_2 \Downarrow v_2 \Rightarrow \Gamma_2 \quad \oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\} \quad (v_1, v_2) \not\leq (\texttt{number}, \texttt{number})}{\Delta; \Gamma \vdash^* e_1 \oplus e_2 \Downarrow \texttt{ArithmeticError} \Rightarrow \emptyset} \text{ (E\_BIN\_OP\_ARITHMETIC\_ERROR2)}$$

$$\frac{\Delta; \Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \quad \Delta; \Gamma \vdash^* e_2 \Downarrow v_2 \Rightarrow \Gamma_2 \quad \oplus \in \{\texttt{div}, \texttt{rem}\} \quad (v_1, v_2) \not\leq (\texttt{integer}, \texttt{integer})}{\Delta; \Gamma \vdash^* e_1 \oplus e_2 \Downarrow \texttt{ArithmeticError} \Rightarrow \emptyset} \text{ (E\_BIN\_OP\_ARITHMETIC\_ERROR3)}$$

$$\frac{\Delta; \Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \quad \Delta; \Gamma \vdash^* e_2 \Downarrow v_2 \Rightarrow \Gamma_2 \quad \oplus \in \{\texttt{and}, \texttt{or}\} \quad v_1 \not\leq \texttt{boolean}}{\Delta; \Gamma \vdash^* e_1 \oplus e_2 \Downarrow \texttt{BadBooleanError} \Rightarrow \emptyset} \text{ (E\_BIN\_OP\_BAD\_BOOLEAN\_ERROR)}$$

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \qquad \mathit{match}(\Gamma, \emptyset, p, v) = \Gamma''}{\Delta; \Gamma \vdash^* p \ = \ e \Downarrow v \Rightarrow \Gamma'} \text{ (E\_MATCH)}$$

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \qquad \mathit{match}(\Gamma, \emptyset, p, v) = \textbf{fail}}{\Delta; \Gamma \vdash^* p \ = \ e \Downarrow \texttt{MatchError} \Rightarrow \emptyset} \text{ (E\_MATCH\_ERROR)}$$

$$\frac{}{\Delta; \Gamma \vdash^* [\,] \Downarrow [\,] \Rightarrow \emptyset} \text{ (E\_ELIST)} \qquad \frac{\Delta; \Gamma \vdash^* e_i \Downarrow v_i \Rightarrow \Gamma_i \quad i = 1, 2}{\Delta; \Gamma \vdash^* [e_1 \,|\, e_2] \Downarrow [v_1 \,|\, v_2] \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (E\_CONS)}$$

$$\frac{\Delta;\Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^* e_n \Downarrow v_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^* \{e_1, \ldots, e_n\} \Downarrow \{v_1, \ldots, v_n\} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (E\_TUPLE)}$$

$$\frac{\Delta;\Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \ \cdots \ \Delta;\Gamma \vdash^* e_n \Downarrow v_n \Rightarrow \Gamma_n}{\Delta;\Gamma \vdash^* \texttt{\%\{}(\overline{k_i \texttt{ => } e_i})_{i\in[n]}\texttt{\}} \Downarrow \texttt{\%\{}(\overline{k_i \texttt{ => } v_i})_{i\in[n]}\texttt{\}} \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (E\_MAP)}$$

$$\frac{\Delta;\Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \qquad \Delta;\Gamma \dagger \Gamma_1 \vdash^* e_2 \Downarrow v_2 \Rightarrow \Gamma_2}{\Delta;\Gamma \vdash^* e_1\texttt{;}e_2 \Downarrow v_2 \Rightarrow \Gamma_1 \dagger \Gamma_2} \text{ (E\_SEQ)}$$

$$\text{(E\_IFE\_TRUE)} \ \frac{\begin{array}{c}\Delta;\Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \quad v \notin \{\texttt{:false}, \texttt{nil}\} \\ \Delta;\Gamma \dagger \Gamma' \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1\end{array}}{\Delta;\Gamma \vdash^* \texttt{if } e \texttt{ do } e_1 \texttt{ else } e_2 \texttt{ end} \Downarrow v_1 \Rightarrow \Gamma'} \qquad \frac{\begin{array}{c}\Delta;\Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \quad v \in \{\texttt{:false}, \texttt{nil}\} \\ \Delta;\Gamma \dagger \Gamma' \vdash^* e_2 \Downarrow v_2 \Rightarrow \Gamma_2\end{array}}{\Delta;\Gamma \vdash^* \texttt{if } e \texttt{ do } e_1 \texttt{ else } e_2 \texttt{ end} \Downarrow v_2 \Rightarrow \Gamma'} \text{ (E\_IFE\_FALSE)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \\ match(\Gamma', \emptyset, p_i, e_i) = \textbf{fail} \quad \forall i \in [1, ..., k-1] \\ match(\Gamma', \emptyset, p_k, e_k) = \Gamma_k \quad \Delta;\Gamma \dagger \Gamma' \dagger \Gamma_k \vdash^* e_k \Downarrow v_k \Rightarrow \Gamma'_k\end{array}}{\Delta;\Gamma \vdash^* \texttt{case } e \texttt{ do } p_1 \texttt{ -> } e_1\texttt{; ...; } p_k \texttt{ -> } e_n \texttt{ end} \Downarrow v_k \Rightarrow \Gamma'} \text{ (E\_CASE)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \\ match(\Gamma', \emptyset, p_i, e_i) = \textbf{fail} \quad \forall i \in [1, ..., n]\end{array}}{\Delta;\Gamma \vdash^* \texttt{case } e \texttt{ do } p_1 \texttt{ -> } e_1\texttt{; ...; } p_k \texttt{ -> } e_n \texttt{ end} \Downarrow \texttt{CaseClauseError} \Rightarrow \Gamma'} \text{ (E\_CASE\_CASE\_CLAUSE\_ERROR)}$$

$$\frac{(f\_name, n) \in dom(\Delta)}{\Delta;\Gamma \vdash^* \&f\_name/n \Downarrow \&f\_name/n \Rightarrow \emptyset} \text{ (E\_ANON)}$$

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = (((p_1^1, \ldots, p_n^1), e^1), \ldots, ((p_1^m, \ldots, p_n^m), e^n)) \\ \Delta;\Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Delta;\Gamma \vdash^* e_n \Downarrow v_n \Rightarrow \Gamma_n \\ match(\emptyset, \emptyset, \{p_1^i, \ldots, p_n^i\}, \{v_1, \ldots, v_n\}) = \textbf{fail} \quad \forall i \in [1, ..., k-1] \\ match(\emptyset, \emptyset, \{p_1^k, \ldots, p_n^k\}, \{v_1, \ldots, v_n\}) = \Gamma^k \\ \Delta;\Gamma^k \vdash^* e^k \Downarrow v \Rightarrow \Gamma'\end{array}}{\Delta;\Gamma \vdash^* f\_name\texttt{(}e_1, \ldots, e_n\texttt{)} \Downarrow v \Rightarrow \Gamma_1 \dagger \cdots \dagger \Gamma_n} \text{ (E\_DCALL)}$$

$$\frac{\begin{array}{c}\Delta[(f\_name, n)] = (((p_1^1, \ldots, p_n^1), e^1), \ldots, ((p_1^m, \ldots, p_n^m), e^n)) \\ \Delta;\Gamma \vdash^* e_1 \Downarrow v_1 \Rightarrow \Gamma_1 \quad \cdots \quad \Delta;\Gamma \vdash^* e_n \Downarrow v_n \Rightarrow \Gamma_n \\ match(\emptyset, \emptyset, \{p_1^i, \ldots, p_n^i\}, \{v_1, \ldots, v_n\}) = \textbf{fail} \quad \forall i \in [1, ..., m]\end{array}}{\Delta;\Gamma \vdash^* f\_name\texttt{(}e_1, \ldots, e_n\texttt{)} \Downarrow v \Rightarrow \emptyset} \text{ (E\_DCALL\_FUNCTION\_CLAUSE\_ERROR)}$$

$$\frac{(f\_name, n) \in dom(\Delta)}{\Delta;\Gamma \vdash^* \&f\_name/n \Downarrow \&f\_name/n \Rightarrow \emptyset} \text{ (E\_ANON)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^* e \Downarrow \&f\_name/n \Rightarrow \Gamma' \\ \Delta;\Gamma \vdash^* f\_name\texttt{(}e_1, \ldots, e_n\texttt{)} \Downarrow v \Rightarrow \Gamma''\end{array}}{\Delta;\Gamma \vdash^* e\texttt{.(}e_1, \ldots, e_n\texttt{)} \Downarrow v \Rightarrow \Gamma' \dagger \Gamma''} \text{ (E\_ACALL)}$$

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow \&f\_name/m \Rightarrow \Gamma' \quad m \neq n}{\Delta; \Gamma \vdash^* e.(e_1, \ldots, e_n) \Downarrow \texttt{BadArityError} \Rightarrow \emptyset} \text{ (E\_ACALL\_BAD\_ARITY\_ERROR)}$$

$$\frac{\Delta; \Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \quad v \neq \&f\_name/n}{\Delta; \Gamma \vdash^* e.(e_1, \ldots, e_n) \Downarrow \texttt{BadFunctionError} \Rightarrow \emptyset} \text{ (E\_ACALL\_BAD\_FUNCTION\_ERROR)}$$

E.3. Algoritmo de match

El algoritmo que define la semántica del *matching* de un valor v respecto a un patrón p de acuerdo a un entorno de valores externo Σ para las variables "pinneadas" y otro como acumulador para las variables del pattern (Γ) es:

procedure MATCH(Σ, Γ, p, v)
  switch (p, v) **do**
    case (l, l)
      return $\emptyset$
    case ($\&f\_name/n, \&f\_name/n$)
      return $\emptyset$
    case ([ ], [ ])
      return $\emptyset$
    case (x, v) **when** $x \notin \Gamma$ **or** $\Gamma[x] = v$
      return $\Gamma[x \mapsto v]$
    case ($\hat{}x, v$) **when** $\Sigma[x] = v$
      return Γ
    case ($[p_1 \mid p_2], [v_1 \mid v_2]$)
      $\Gamma \leftarrow$ MATCH(Σ, Γ, p_1, v_1)
      return MATCH(Σ, Γ, p_2, v_2)
    case ($\{p_1, ..., p_n\}, \{v_1, ..., v_n\}$)
      for $i \in [1, ..., n]$ **do**
        $\Gamma \leftarrow$ MATCH(Σ, Γ, p_i, v_i)
      end for
      return Γ
    case (%{$\overline{(k_i \Rightarrow p_i)_{i \in I}}$}, %{$\overline{(k_i \Rightarrow v_i)_{i \in J}}$}) **when** SUBSET($I, J$)
      for $i \in I$ **do**
        $\Gamma \leftarrow$ MATCH(Σ, Γ, p_i, v_i)
      end for
      return Γ
    default
      fail
end procedure

E.4. Reglas de evaluación de los casts

$$\frac{\Delta;\Gamma \vdash^* e \Downarrow v \Rightarrow \Gamma' \qquad \Delta;\Gamma \dagger \Gamma' \vdash^* (v \mid \tau \leadsto \sigma) \Downarrow v' \Rightarrow \Gamma''}{\Delta;\Gamma \vdash^* (e \mid \tau \leadsto \sigma) \Downarrow v' \Rightarrow \Gamma' \dagger \Gamma''} \text{ (E\_CAST)}$$

$$\frac{type(l) \leq \tau}{\Delta;\Gamma \vdash^* (l \mid \tau \leadsto \tau) \Downarrow l \Rightarrow \Gamma} \text{ (E\_CAST\_LIT)} \qquad \frac{[\,] \leq \tau_1 \quad [\,] \leq \tau_2 \quad \tau_1 \sim \tau_2}{\Delta;\Gamma \vdash^* ([\,] \mid \tau_1 \leadsto \tau_2) \Downarrow [\,] \Rightarrow \Gamma} \text{ (E\_CAST\_ELIST)}$$

$$\frac{\Delta;\Gamma \vdash^* [(v_1 \mid \tau \leadsto \sigma) \mid (v_2 \mid [\tau] \leadsto [\sigma])] \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* ([v_1 \mid v_2] \mid [\tau] \leadsto [\sigma]) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_LIST)}$$

$$\frac{\Delta;\Gamma \vdash^* \{(v_1 \mid \tau_1 \leadsto \sigma_1), \ldots, (v_n \mid \tau_n \leadsto \sigma_n)\} \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (\{v_1, \ldots, v_n\} \mid \{\tau_1, \ldots, \tau_n\} \leadsto \{\sigma_1, \ldots, \sigma_n\}) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_TUPLE)}$$

$$\frac{J \subseteq I \qquad w_j = (v_j \mid \tau_j \leadsto \sigma_j) \quad \forall j \in J \qquad w_i = v_i \quad \forall i \in I \setminus J \qquad \Delta;\Gamma \vdash^* \%\{(\overline{k_i \Rightarrow w_i})_{i \in I}\} \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (\%\{(\overline{k_i \Rightarrow v_i})_{i \in I}\} \mid \%\{(\overline{k_i \Rightarrow \tau_i})_{i \in J}\} \leadsto \%\{(\overline{k_i \Rightarrow \sigma_i})_{i \in J}\}) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_MAP)}$$

$$\frac{\Delta;\Gamma \vdash^* (v.((v_1 \mid \sigma_1 \leadsto \tau_1), \ldots, (v_n \mid \sigma_n \leadsto \tau_n)) \mid \tau_0 \leadsto \sigma_0) \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (v \mid (\tau_1, .., \tau_n) \to \tau_0 \leadsto (\sigma_1, .., \sigma_n) \to \sigma_0).(v_1, \ldots, v_n) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_CALL)}$$

$$\frac{\Delta;\Gamma \vdash^* v \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (v \mid \texttt{any} \leadsto \texttt{any}) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_ANY\_ANY)}$$

$$\frac{\tau \neq \texttt{any} \qquad \Delta;\Gamma \vdash^* (v \mid \tau \leadsto ground_T(\tau)) \Downarrow r \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (v \mid \tau \leadsto \texttt{any}) \Downarrow r \Rightarrow \Gamma'} \text{ (E\_CAST\_GROUND\_ANY)}$$

$$\frac{\tau \neq \texttt{any} \quad ground_V(v) \lesssim ground_T(\tau) \qquad \Delta;\Gamma' \vdash^* (v \mid ground_V(\tau) \leadsto \tau) \Downarrow w \Rightarrow \Gamma'}{\Delta;\Gamma \vdash^* (v \mid \texttt{any} \leadsto \tau) \Downarrow w \Rightarrow \Gamma'} \text{ (E\_CAST\_ANY\_GROUND)}$$

$$\frac{\sigma \neq \texttt{any} \quad ground_V(v) \not\lesssim ground_T(\sigma)}{\Delta;\Gamma \vdash^* (v \mid \texttt{any} \leadsto \sigma) \Downarrow \texttt{CastError} \Rightarrow \emptyset} \text{ (E\_CAST\_CAST\_ERROR)}$$

$$\frac{ground_V(v) \not\lesssim ground_T(\tau) \quad \text{ó} \quad \tau \nsim \sigma}{\Delta;\Gamma \vdash^* (v \mid \tau \leadsto \sigma) \Downarrow \texttt{BadCastError} \Rightarrow \emptyset} \text{ (E\_BAD\_CAST\_ERROR)}$$

siendo que las funciones $ground_V$ y $ground_T$ están definidos por los siguientes procedimientos:

```
procedure GroundV(v)
    switch v do
        case l : return Type(l)
        case [ ] : return [ ]
        case [v] : return [any]
        case {v_1 , ... , v_n}
            return {any , ... , any}
        case %{k_1 => v_1 , ... , k_n => v_n}
            return %{k_1 => any , ... , k_n => any}
        case &f_name/n
            return (any, .., any_n) → any
        case (v | (τ_1, .., τ_n) → τ_0 ~> (σ_1, .., σ_n) → σ_0)
            return (any, .., any_n) → any
        default : fail
end procedure
```

```
procedure GroundT(τ)
    switch τ do
        case b when b ∈ B : return b
        case [ ] : return [ ]
        case [τ] : return [any]
        case {τ_1 , ... , τ_n}
            return {any , ... , any}
        case %{k_1 => τ_1 , ... , k_n => τ_n}
            return %{k_1 => any , ... , k_n => any}
        case (τ_1, .., τ_n) → τ_0
            return (any, .., any_n) → any
        case any
            return any
end procedure
```