UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

EXTENSIÓN DE UN SISTEMA DE TIPOS GRADUAL PARA REGISTROS USANDO TIPOS UNIÓN

T E S I S

QUE PARA OPTAR POR EL GRADO DE

DOCTORA EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

KARLA RAMÍREZ PULIDO

TUTOR PRINCIPAL:

Dr. Jorge Luis Ortega Arjona, Facultad de Ciencias, UNAM.

COMITÉ TUTOR:

Dr. Francisco Hernández Quiroz, Facultad de Ciencias, UNAM.

Dr. Favio Ezequiel Miranda Perea, Facultad de Ciencias, UNAM.

CIUDAD UNIVERSITARIA, CD. MX. ABRIL DE 2022

Dedicatoria

A mi mamá Ana María (Q.E.P.D.) quien ha sido un ejemplo de perseverancia y a quien quiero y extrañaré toda mi vida. A mi papá Juan Ernesto quien ha confiado en mi y me ha dado sus consejos para seguir, aún y cuando creía que ya no se podía más. A ambos: por quererme así, tal cual soy.

Agradecimientos

A mi tutor, el Dr. Jorge Luis Ortega Arjona por todo su apoyo, paciencia y tiempo durante la realización de esta investigación, ya que sin su guía este trabajo no hubiera sido posible. Por esos consejos que, tanto en el ámbito académico como profesional y personal me permitieron seguir adelante, en verdad muchas gracias Jorge por todo lo que mas apoyado y por darme la oportunidad de aprender de ti. A mi Comité Tutoral, el Dr. Francisco Hernández Quiroz y el Dr. Favio Miranda Perea, por todo el tiempo que dedicaron a las revisiones de este trabajo doctoral, por las experiencias y conocimientos compartidos: muchas gracias. Así como a mi jurado completo, la Dra. Lourdes del Carmen González Huesca y el Dr. Ismael Everardo Bárcenas Patiño quienes compartieron sus sugerencias, conocimientos y tiempo para hacer de éste un mejor trabajo de investigación.

Muy particularmente agradezco (infinitamente) a Lulú González Huesca por todo el tiempo, esfuerzo, y experiencias (personales y académicas) que compartió conmigo al realizar este trabajo, así como los artículos que se generaron de éste, en verdad muchas muchas gracias, sin tu guía y acompañamiento esto no hubiera sido posible de desglozar y concretar.

A mi familia y amigos que han estado conmigo dándome ánimos y echandome porras para que siguiera adelante en hacer realidad este sueño que tenía desde hace mucho tiempo. A mi mamá Ana María (Q.E.P.D.) quien siempre ha sido un ejemplo a seguir y a quien extrañaré toda mi vida, a mi papá Juan Ernesto que me ha brindado sus consejos para seguir adelante y ha compartido su tiempo y enseñanzas, a mis hermanos Pao y Ado, porque son los mejores siempre, en las buenas y en las malas, y no me han dejado caer. A Sergio Padilla muy especialmente por su apoyo incondicional y porque me ha hecho ver la vida un poco diferente, así más feliz y más bonita. Y por supuesto a Cheiser pug, porque es un excelente escucha y me levanta el ánimo con su compañía, aún en los momentos de más desesperación.

Y por último pero no menos importante, a mi querida Universidad Nacional Autónoma de México, por todo lo que me ha dado todo este tiempo, por ser mi segunda casa.

Contenido

Índice de figuras

Índice de tablas

Índice de códigos

Capítulo 1

Introducción

> "Si supiese qué es lo que estoy haciendo, no le llamaría investigación, ¿cierto?"
>
> *Albert Einstein.*

Contexto

Una posible categorización de Lenguajes de Programación está dada por el estilo de tipificación que éstos presentan: estático y dinámico [56, 64, 66]. Por un lado, el análisis sintáctico y la verificación de tipos aseguran que esté bien formado un programa de manera estática; mientras que la fase dinámica debe de garantizar la consistencia de los tipos en un programa en tiempo de ejecución. Es decir, tanto el tipificado estático como dinámico de un programa deben de mantenerse consistentes [47, 66, 78, 79].

Tanto los sistemas de verificación de tipos estáticos como los dinámicos presentan diferentes características. Cada uno provee diversos mecanismos para asegurar su consistencia de manera más eficiente y segura. Por un lado, los lenguajes con tipificado estático (1) ayudan al programador a autodocumentar el programa con solo tener etiquetados los tipos de los identificadores utilizados, (2) ayudan a tener una mayor seguridad en cuanto al uso de ciertas funciones en un programa, (3) mejoran el rendimiento de un programa en tiempo de ejecución, ya que desde tiempo de compilación el programador define los tipos utilizados dentro de éste, por lo que se pueden minimizar los errores asociados a los tipos, (4) además se preserva la consistencia de los tipos de valores que regresará un programa en tiempo de ejecución. Por otro lado, el tipificado dinámico permite a los programadores no utilizar anotaciones explícitas asociadas a los tipos, lo cual representa una ventaja al momento de desarrollar un sistema, pues le permite al programador (1) tener una mayor flexibilidad al momento de programar, (2) facilitar la implementación rápida de prototipos, los cuales son a su vez (3) fáciles de probar [56, 61, 78, 79, 87, 105].

En los últimos años se han propuesto diferentes enfoques que mezclan tanto las características del tipificado estático como del dinámico [10, 39, 93]. Por ejemplo, tanto la seguridad como el rendimiento del tipificado dinámico se pueden mejorar agregando opcionalmente anotaciones de tipo [10, 39, 79, 93]. Dentro de los lenguajes con tipificado dinámico se encuentran los lenguajes con tipificado gradual en el cual el programador tiene la opción de tipificar o no una expresión de manera explícita. Otro tipo de lenguajes donde se mezcla el tipificado estático y dinámico son los de tipificado *suave* (traducción del inglés del término *soft typing*), que pueden realizar inferencia de tipos, a partir de las operaciones básicas o primitivas del lenguaje. Otros tipos de lenguajes son los que utilizan un sistema de tipos nominal, en donde se permite tener anotaciones de tipos de manera opcional (también conocido como *duck typing* [58]) dentro del estilo de la Programación Orientada a Objetos [58, 78].

Independientemente de la clasificación a la que pertenezca un lenguaje de programación, éstos son utilizados por los programadores para resolver alguna necesidad o problema en cualquier ámbito. Con el creciente uso de las tecnologías y por ende de sistemas computacionales escritos en algún o algunos lenguajes de programación, el desarrollo de software se ha vuelto una tarea cada vez más compleja, por lo que dar una solución óptima

al problema que se desee resolver, no solo depende del programador sino de la herramienta (lenguaje de programación) que se haya elegido. Por lo descrito anteriormente, los diseñadores de lenguajes estudian y prueban nuevos enfoques en el desarrollo de sistemas de tipos que aseguren la consistencia de un programa tanto en tiempo de compilación como de ejecución [64, 106].

Siguiendo esta idea de brindar a los programadores más construcciones en el lenguaje, se propone el siguiente problema a investigar.

Problema propuesto

Se propone extender un lenguaje dentro del paradigma funcional con tipificado gradual para que pueda manejar registros (*records*), preservando en ambos casos las propiedades de seguridad y consistencia de un lenguaje gradual.

Krishnamurthi en [56] define la seguridad como ninguna operación primitiva deberá ser aplicada a valores de tipo incorrecto, y la propiedad de consistencia como para todo programa p escrito en un lenguaje de programación, el programa termina con un valor v tal que $v : T$ o bien levantá una excepción de un conjunto de excepciones bien definida [1]. La propiedad de seguridad en un sistema de tipos según Pierce [66] indica que los términos bien tipificados no lleguen a un estado de estancamiento, esto no se refiere a que sea un valor final, sino a que las reglas de evaluación no nos indiquen qué hacer a continuación. Regularmente, esto se prueba usando los teoremas de Progreso y Preservación. El primer teorema (Progreso) nos indica si un término bien tipificado no está estancado (ya que es un valor o se puede realizar un siguiente paso, de acuerdo a las reglas de evaluación), y el segundo teorema (Preservación) nos indica que si un término bien tipificado puede tomar un paso de la evaluación entonces el término resultante estará bien tipificado también.

Lo anterior resulta útil a la comunidad de lenguajes de programación, pues al extender tal lenguaje con registros se provee un lenguaje más completo, añadiendo expresividad a través de estas construcciones y sus respectivas funcionalidades. Así mismo, se sientan las bases para su estudio formal a través de demostraciones para asegurar la consistencia del lenguaje extendido con este tipo de estructuras de datos. Cabe mencionar que este trabajo de tesis se realiza en el marco de la teoría de lenguajes de programación.

Solución propuesta

Esta investigación se centra en extender la expresividad de un lenguaje funcional gradual usando las construcciones de registros, analizando la semántica estática y dinámica de un lenguaje con tipificado gradual, garantizando la consistencia y seguridad de éstos.

Agregar nuevas características a un lenguaje de programación gradual permite enriquecerlo. Esta construcción resulta interesante en el diseño y uso de lenguajes, pues es un constructor deseable en lenguajes actuales con tipificado estático o dinámico. Esta extensión se realiza preservando formalmente la propiedad de consistencia y seguridad.

Hipótesis

Se propone la siguiente pregunta de investigación en este trabajo de tesis:

> ¿Es posible mantener la consistencia[2] para todo programa en un lenguaje diseñado bajo un sistema de tipos gradual al extenderlo con registros usando el enfoque de tipos unión?

[1] Algunos otros autores contemplan la propiedad de seguridad o solidez como la propiedad principal a demostrarse en lenguajes de programación, ya que ésta comprende la demostración formal de los Teoremas de Progreso y Preservación.

[2] En este trabajo la propiedad de consistencia también traducida como solidez o corrección, del inglés *soundness*, la cual se refiere a que el verificador de tipos en un sistema de tipos, pueda garantizar las propiedades de progreso y preservación [56].

Objetivos

I. Objetivo principal:

- Extender un sistema de tipos gradual con registros usando tipos unión de manera consistente y segura.

II. Objetivos secundarios:

- Definir un sistema de tipos gradual para soportar registros.
- Derivar la semántica estática y dinámica del lenguaje gradual usando tipos unión, preservando y probando las características en cada uno de los lenguajes utilizados: $STFL$, $GTFL$ y $GTFL_{\oplus}$.
- Demostrar que se mantienen las propiedades de seguridad y consistencia en cada uno de los lenguajes al añadir registros.
- Demostrar que $GTFL_{\oplus}$ preserva las propiedades de garantías graduales estáticas y dinámicas graduales, seguridad de tipos y una equivalencia entre las referencias (basadas en evidencias) y la semántica traslacional.
- Demostrar que se preserva la operación de conversión de tipos (*cast*), y la relación lógica siguiendo la interpretación de la GUT, basada en la compilación de un Cálculo *Threesome*.

Metodología

En la presente tesis doctoral se propone utilizar el Tipificado Gradual de Tipos Unión diseñada por Toro y Tanter [95], basada a su vez en el Tipificado Gradual Abstracto para probar formalmente su consistencia y seguridad. Esto implica que el sistema de verificación gradual debe ser compatible con el sistema de tipos original propuesto por Toro y Tanter, es decir, preservando la consistencia de tipos, tanto de forma estática como dinámica al extenderlo con registros.

Para extender la gramática de manera consistente y segura se proponen los siguientes pasos entre lenguajes:

1. Extender la gramática STFL, para que maneje registros.
2. Extender la gramática GTFL, para que maneje registros.
3. Extender la gramática $\text{GTFL}_{\oplus}$, para que maneje registros.
4. Extender la relación lógica para manejar registros en el lenguaje intermedio.

La Figura 1.1 muestra las transformaciones entre STFL y GTFL usando la función $\alpha_?$ para abstraer, y la función $\gamma_?$ para concretizar. El siguiente paso es convertir el lenguaje GTFL al lenguaje $\text{GTFL}_{\oplus}$. haciendo uso de la función de abstracción $\alpha_{\oplus}$, y la función de concretización $\gamma_{\oplus}$ [95].

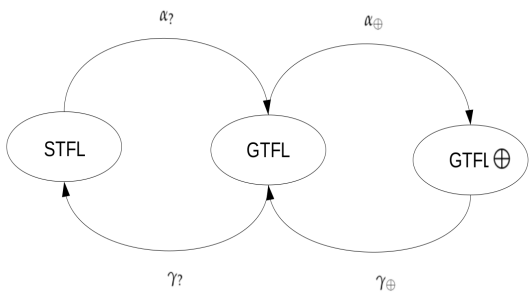

Figura 1.1: Metodología propuesta por Toro y Tanter, usando las funciones $\alpha_?$, $\alpha_{\oplus}$, $\gamma_?$, y $\gamma_{\oplus}$ para convertir el lenguaje STFL en el lenguaje $GTFL_{\oplus}$.

La extensión propuesta para registros está basada en la metodología propuesta por Toro y Tanter [95]. La Figura 1.1 muestra las transformaciones requeridas para obtener la extensión para registros en $\text{GTFL}_{\oplus}$. El primer paso es definir y probar formalmente, tanto la función de abstracción $\alpha_?$ del lenguaje STFL al lenguaje GTFL, así como la función de concretización $\gamma_?$ entre el lenguaje GTFL y STFL. El segundo paso es obtener las transformaciones respectivas entre el lenguaje $\text{GTFL}_{\oplus}$ al lenguaje GTFL, usando las funciones $\alpha_{\oplus}$ y $\gamma_{\oplus}$ entre ambos lenguajes.

Proponer las reglas de conversión de tipos para que pueda ser utilizada usando el Cálculo *Threesome*, así como las relaciones lógicas propias para registros y demostrar su corrección en el lenguaje gradual.

La metodología de Toro y Tanter [95] está a su vez basada en la metodología AGT, en la cual la semántica de los tipos graduales se encuentra compuesta por dos distintas interpretaciones abstractas: uniones de tipos y tipos graduales, es decir uniones graduales de tipos. Esta metodología a su vez resulta ser interesante debido a que (a) combina los beneficiones de las uniones con etiquetas y sin etiquetas, añadiendo cierta flexibilidad estática respaldada en las verificaciones en tiempo de ejecución. Lo anterior es diferente a si solo se tuviera la metodología de tipos graduales donde se puede rechazar ciertos programas de manera estática desde un inicio; (b) se utiliza la AGT estratificada para derivar la semántica estática del lenguaje gradual, haciendo uso de conexiones de Galois entre los tipos graduales y los conjuntos de tipos estáticos, las cuales permiten el levantamiento de funciones y predicados sobre tipos estáticos y los tipos graduales; (c) al usar la metodología AGT para formalizar esta metodología demostrando tanto la propiedad de seguridad como de las garantías graduales de tipos propuestas por Siek et al. [82] se puede concluir con los resultados obtenidos que el lenguaje, después de todas las transformaciones que se tienen, es seguro y consistente; y (d) por último, al tener las transformaciones a un lenguaje intermedio usando el Cálculo Threesome, se tiene una representación eficiente en manejo de espacio para utilizar la conversión de tipos. Se demuestra también la corrección de la compilación de éste usando las relaciones lógicas propuestas.

Contribuciones

El principal objetivo de esta tesis es integrar un sistema de tipos que soporte el tipificado gradual de un lenguaje funcional, el cual sea consistente con las reglas de tipado estático y dinámico. Las principales contribuciones de esta investigación son las siguientes:

- Reunir y dar una propuesta de los diferentes enfoques que existen en relación a las diferentes clasificaciones de lenguajes de programación dados sus tipos.
- Analizar y diseñar de un sistema de tipos basado en el tipificado gradual de un lenguaje funcional.
- Extender un sistema de tipos gradual con registros, preservando la seguridad y consistencia de éste durante todas las transformaciones que implica su desarrollo.
- Demostrar que el tipificado gradual puede ser extendible a otros lenguajes funcionales con las mismas características que el evaluado en este trabajo, es decir, usando estructuras de datos con comportamientos similares a la de los registros.

Cabe mencionar que se han hecho algunos cambios al trabajo realizado por Toro y Tanter en [95, 96]. Estos cambios son necesarios pues se encontraron inconsistencias en dichas publicaciones, por ejemplo en varias defincions, proposiciones, y pruebas formales expuestas en los trabajos antes mencionados. Este trabajo de tesis doctoral contiene tanto los cambios como las omisiones de definiciones en los trabajos antes mencionados.

La motivación principal de este trabajo es exponer el enfoque GUT estudiando un lenguaje funcional con registros, siguiendo la propuesta de [95, 96], quienes hacen uso de la metodología de [42]. La principal aportación de este trabajo es la revisión exhaustiva del GUT mediante el estudio del caso de una extensión tradicional[3] del lenguaje STFL (*Simply-Typed Lambda Calculus*) para registros, mostrando que el lenguaje resultante, es decir el lenguaje gradual preserva las propiedades graduales tanto estática como dinámicamente. Esta contribución es

[3]Por tradicional nos referimos a un lenguaje sin subtipificado. La propiedad denominada extensión conservadora por [25] no está relacionada con la extensión presentada en este trabajo.

una de las primeras extensiones de un lenguaje gradual con tipos unión para este tipo de estructuras de datos, utilizándola a su vez para entender a detalle el marco GUT.

Por lo anterior, el objetivo de este trabajo es proponer un lenguaje con tipificado gradual incluyendo registros, preservando la seguridad de tipos junto con garantías graduales. Estas garantías son estándar para su semántica estática y semántica dinámica [82]. El principal reto a resolver es mantener la consistencia de los tipos en el lenguaje, incluso cuando el tipo desconocido (unknown) está presente, utilizando la metodología desarrollada por Toro y Tanter. Por lo que, la metodología AGT se añade para reconstruir las nociones relacionadas con la tipificación gradual. El uso de la metodología AGT ayuda a diseñar sistemas de tipos graduales, y en este caso, cuando se añaden registros al lenguaje, se mantienen tanto la seguridad como las garantías graduales.

Vale la pena aclarar que hay al menos dos formas de añadir registros a un lenguaje: los registros tal y como se han discutido en este trabajo doctoral (que llamamos conservador), y la extensión que proporciona al diseñar filas graduales (*gradual raws*), el cual se introduce como un nuevo tipo gradual, que representan campos desconocidos (unknown) extra en un registro. A su vez, otra diferencia entre [42] y el trabajo presentado por García et al. es que en éste se utiliza el subtipificado consistente y en este trabajo doctoral no se considera la subtipificación.

La propuesta se centra en el estudio de registros tradicionales donde el tamaño, y el orden de los campos tienen que ser los mismos, con el fin de comparar dos o más registros. Esto permite analizar y exponer el enfoque de Toro y Tanter, en el que se utiliza el AGT para construir un lenguaje gradual incremental, probando estructuralmente sus propiedades, principalmente para preservar la seguridad de cada uno lenguajes.

Sin embargo, resulta complicado garantizar la consistencia de un programa con Tipificado de Unión Gradual, ya que no todas las expresiones tienen definidos todos sus tipos, pues el tipo desconocido (?) está presente, una vez que se inicia el desarrollo de este tipo de metodología. Por ejemplo, si se considera la función de abstracción aplicada a los siguientes ejemplos con registros se tiene lo siguiente [42]:

- $\alpha([\ell_1 : Bool]) = [\ell_1 : Bool]$
- $\alpha([\ell_1 : Bool], [\ell_1 : Int]) = [\ell_1 : ?]$
- $\alpha([\ell_1 : Int], [\ell_1 : Int, \ell_2 : Bool]) = ?$

El primer caso se refiere a un registro de una única etiqueta (ℓ_1) asociada al tipo booleano ($Bool$); el resultado entonces tiene el mismo tipo que el declarado en el registro ($Bool$). El segundo ejemplo muestra dos registros con la misma etiqueta (ℓ_1) pero con tipos diferentes ($Bool$ and Int). En este caso, la abstracción pierde cierto tipo de información, ya que el verificador de tipos no puede deducir el tipo de la etiqueta ℓ_1 cuando ésta sea utilizada. En el tercer ejemplo, se muestran dos registros con diferente estructura pero que comparten la misma etiqueta, es decir, tienen el mismo nombre (ℓ_1), por lo que el verificador de tipos no puede decidir a cuál tipo se refiere el programador, cuando éste los utilice.

Cabe mencionar que se define una función de abstracción, que representa la colección de tipos estáticos con la mayor precisión posible en términos de un tipo gradual [42]. En los dos últimos ejemplos anteriores, el resultado de aplicar la función de abstracción a éstos no puede determinar el tipo de la etiqueta a la cual se hace referencia. En el segundo caso, como se tienen dos registros cuyos nombres de etiqueta son el mismo, al acceder al mismo identificador de la primer etiqueta la función de abstracción no podrá determinar el tipo de ésta. En el tercer caso los registros son distintos en el número de elementos que contienen, sin embargo en ambos está presente el mismo identificador para la primera etiqueta, por lo que al aplicar la función de abstracción resulta imposible conocer a cuál de los registros se podría estar haciendo referencia, pues en ambos figura el mismo identificador. En otras palabras, la función de abstracción desconocerá el tipo que podrá abstraer y así determinar la unión de tipos tomando en cuenta el tipificado gradual.

Este trabajo extiende la propuesta de Toro y Tanter para manejo de registros, lo cual resulta interesante pues nos permite analizar y demostrar las propiedades de seguridad y garantías graduales de un lenguaje con tipificado gradual que presente este tipo de estructuras de datos o estructuras con un comportamiento similar, como son los **structs** de Racket, arreglos y tuplas (estructuras presentes en muchos lenguajes de programación como parte de su núcleo), es decir, estructuras donde se pueda acceder a sus elementos por medio de una etiqueta o un índice. Lo importante en este tipo de propuesta radica en las demostraciones que se tienen que realizar en cada una de las transformaciones para probar que tanto la semántica estática como dinámica mantienen la seguridad del lenguaje, al incluir este tipo de estructuras.

Cabe mencionar que la propuesta en la que se basa este trabajo (Toro y Tanter [95]) difiere de la propuesta hecha por Castagna et al. [20] principalmente en que se utiliza el subtipificado polimórfico y polimorfismo paramétrico implícito, y en este trabajo no se utiliza el subtipificado, y aunque el siguiente paso en la investigación es el añadir variables polimórficas, se tendría que seguir la misma metodología aquí utilizada, la cual no conlleva el uso de subtipificado, no así el uso de uniones graduales.

Estructura de la tesis

Este trabajo de tesis está estructurado de la siguiente manera:

- **Capítulo 2. Antecedentes.** En este capítulo se provee una introducción al área de lenguajes de programación, así como algunas de sus clasificaciones dependiendo de diversos criterios. A su vez se provee una introducción a los conceptos relacionados con el tipificado dinámico y estático, así como las ventajas y desventajas de dichas categorizaciones. Asimismo, se provee una descripción de los diferentes enfoques relacionados con el tipificado gradual en lenguajes de programación, introduciendo el vocabulario usado en el presente trabajo, el cual es utilizado y aplicado en los siguientes capítulos para describir conceptos relacionados con el Tipificado Gradual, (TG).

- **Capítulo 3. Trabajo relacionado.** En este capítulo se introduce una revisión de algunos de los trabajos más relevantes relacionados con el TG. Este capítulo está organizado como sigue: primero una revisión de los trabajos relacionados con los lenguajes con tipificado gradual describiendo algunas nociones básicas del área. Posteriormente se presentan los principales enfoques relacionados con la tipificación gradual, describiendo brevemente sus características. Y por último se presenta una sección con los principales trabajos relacionados con el área.

- **Capítulo 4. Extensión del lenguaje con Tipificación Gradual usando Registros.** Este capítulo presenta la Tipificación Gradual como una especificación formal que ayuda a determinar los tipos de una expresión en el lenguaje y su consistencia tanto en tiempo de compilación como en tiempo de ejecución. De esta manera este capítulo presenta un sistema de tipificado gradual para un lenguaje funcional extendido con registros. En este estudio se contemplan como tipos primitivos los enteros, booleanos, funciones y registros (en el primer lenguaje). Posteriormente cuando se hace uso de los tipos graduales se contempla el tipo desconocido ?, además de los tipos anteriores, así como las uniones graduales $T1 \oplus T2$. Siguiendo el enfoque de conjuntos de tipos estáticos (en el sentido de la interpretación abstracta) utilizado en el Tipificado Gradual Abstracto (del inglés, AGT) [42]. En particular, el tipo ? abstrae cualquier tipo posible y la unión $T1 \oplus T2$ abstrae tanto a $T1$ como a $T2$. Este trabajo explota el enfoque estratificado de AGT utilizando la metodología de Toro y Tanter [95] para tratar la unión gradual. En otras palabras, se extiende el lenguage *Simply Typed Functional Language* con registros, STFL_{rec}. Posteriormente se introduce el lenguaje GTFL_{rec} con su sistema de tipos estáticos y semántica operacional, y los tipos graduales se introducen siguiendo el enfoque estratificado de [95] para la unión de tipos y el tipo desconocido ?. Posteriomente, se introduce al lenguaje con tipificado gradual de $\mathsf{GTFL}^{\oplus}_{rec}$ para manejar una versión con tipos intrínsecos que permite la comprobación de los términos bien tipificados, junto con su semántica dinámica. Para finalmente hacer la traducción consistente a un lenguaje intermedio con conversiones de tipo que están bien definidas utilizando relaciones lógicas.

  Es importante resaltar que durante la realización de esta tesis doctoral se corrijieron varios errores del trabajo original propuesto por Toro y Tanter [95] y [96], como la relación de persistencia gradual, algunas demostraciones, y ejemplos de uso expuestos en [96]. A su vez, se propusieron algunas definiciones formales (como la definición y relación de evidencias) para así un trabajo de investigación más completo.

- **Capítulo 5. Conclusiones.** En este capítulo se presenta una síntesis de la investigación, bajo un enfoque de resumen crítico de la investigación *per se*, haciendo referencia a la hipótesis y a las contribuciones de la investigación. También se presenta una sección de trabajo a futuro, en la cual se resumen las líneas a futuro del trabajo de investigación que se pueden analizar a partir de este trabajo.

Capítulo 2

Antecedentes

> "El diseño de lenguajes de programación es como pasear en el parque. Bueno, en parque jurásico."
>
> *Larry Wall* [97].

El objetivo de este capítulo es proveer (a) una introducción al área de lenguajes de programación, explicando de manera general las características de éstos y algunas de sus clasificaciones dependiendo de diversos criterios; (b) una introducción a los lenguajes con tipificado estático y dinámico, así como las ventajas y desventajas que presentan este tipo de lenguajes dependiendo de la clasificación en la que se encuentren; y (c) una introducción a los lenguajes con tipificado gradual, introduciendo el vocabulario que se presenta en este trabajo de investigación, el cual es usado en el resto de los capítulos.

Introducción a los lenguajes de programación

Existen varias acepciones del concepto de lenguaje de programación definidas por varios autores, hasta el momento. A continuación se exponen algunos de los conceptos encontrados, los cuales resultan los más adecuados para fines de este trabajo:

Según Harper, los lenguajes de programación permiten expresar cálculos que pueden ser procesados y entendidos tanto por las computadoras como por los programadores [47].

Un lenguaje de programación es un lenguaje formal, el cual es [73]:

> "Un conjunto definido de manera recursiva, de cadenas de caracteres en un alfabeto".

Y el término alfabeto según Martin Davis y Elaine Ewyuker se define como [32]:

> "Un conjunto finito y no vacío de símbolos o letras".

Todos los lenguajes de programación deben de contar con cierta *sintaxis*, *semántica*, *bibliotecas* e *idioms* o modismos [56]. La sintaxis se refiere a la forma en la que un programa debe ser escrito, es decir, cómo se deben de escribir los enunciados, declaraciones, y otras construcciones del lenguaje [68]. Cada regla tiene asociada cierta semántica, es decir, su significado. La semántica que provee cada enunciado en un programa permite al programador establecer un significado a cada expresión en particular y a un programa en general. Las bibliotecas se definen como el conjunto de funciones previstas por el lenguaje para ser usadas por los programadores. Y finalmente los *idioms* que son patrones de bajo nivel definidos para cada lenguaje de programación [17].

Proveer una definición de **lenguajes de programación** es difícil, sin embargo, tomando en cuenta las definiciones provistas por los autores antes citados, y para los propósitos de este trabajo de tesis se proporciona una definición dentro de este contexto:

> Un lenguaje es un conjunto de cadenas sobre un alfabeto y un alfabeto es un conjunto de símbolos. Finalmente un programa es una cadena que se busca decidir si está bien escrita siguiendo las reglas del lenguaje y de forma paralela se puede analizar semánticamente.

Para que un código fuente (programa computacional) se pueda transformar en código ejecutable, es decir, se pueda ejecutar en una máquina, se deben de realizar varios tipos de análisis [6]:

Análisis léxico. Descompone la entrada (código fuente) en componentes léxicos también conocidos como símbolos (del inglés *tokens*), es decir, analiza la secuencia de caracteres, palabras reservadas y marcas de puntuación de un programa, quitando los espacios en blanco y comentarios.

Análisis sintáctico. Analiza sintácticamente las estructuras de un programa, produciendo un árbol de sintaxis abstracta.

Análisis semántico. Atribuye un significado a cada estructura de un programa, verificando que cada expresión sea del tipo correcto y traduce la sintaxis abstracta en una representación adecuada para generar código de máquina ejecutable.

Todo código fuente pasa por diversas etapas para poder generar código ejecutable. Tal transformación se genera, ya sea por medio de un compilador o bien de un intérprete, dependiendo si el lenguaje es compilado o interpretado. Durante esta conversión, los programas escritos en un lenguaje de programación de alto nivel pueden estar tipificados explícita o implícitamente dada su sintaxis concreta.

Un programa escrito en una sintaxis concreta transforma ésta en una sintaxis abstracta. Esta última es ejecutada por un compilador o un intérprete de algún lenguaje de programación en alguna computadora [38].

La Figura 2.1 es una representación de los tipos de análisis que se generan para obtener un código ejecutable a partir del código escrito por un programador. En tal figura se pueden observar cinco bloques numerados. El bloque número 1 denota al análisis léxico. Dicho bloque se encarga de recibir el código fuente del programador. Este tipo de análisis separa cada parte del programa en componentes léxicos o símbolos (*tokens* en inglés).

Un componente léxico[1] está definido como [6]:

> "Una secuencia de caracteres que pueden ser tratados como una unidad en la gramática de un lenguaje de programación".

Una vez identificados todos los componentes léxicos que contiene un código fuente, el análisis sintáctico (ver el bloque 2 de la Figura 2.1) se encarga de verificar que cada símbolo sea parte de alguna construcción sintáctica del lenguaje de acuerdo a las reglas de sintaxis del mismo. Este análisis genera un árbol de sintaxis abstracta; este árbol es la entrada de lo que recibe el análisis semántico (ver bloque 3 de la Figura 2.1) el cual se encarga de asociarle un significado a cada construcción sintáctica generada en el árbol de sintaxis abstracta, generando así código intermedio ejecutable; posteriormente en el bloque 4 se generan las optimizaciones de código respectivas para obtener un código más eficiente, estas optimizaciones generan entonces un código optimizado; una vez que el código está optimizado se realiza la fase de ligado (bloque 5), en la cual se vincula el código objeto de otras aplicaciones con el código generado por el código fuente para finalmente generar código ejecutable.

Lo descrito anteriormente es una representación básica de las fases por las que pasa un lenguaje de programación el cual está constituido por símbolos dentro de una gramática, la cual es a su vez analizada sintácticamente y traducida a una forma ejecutable, de acuerdo a las reglas semánticas del lenguaje. Esta representación recibe una construcción de entrada la cual tiene su respectiva representación de salida. Lo descrito en un principio conlleva una especificación inicial que genera un especificación ejecutable [22]. Cabe mencionar que existen más

[1] Los componentes léxicos tales como `if, void, return`, entre otros, son construcciones de caracteres llamados palabras reservadas, y en muchos lenguajes no se pueden usar como identificadores por los programadores [6].

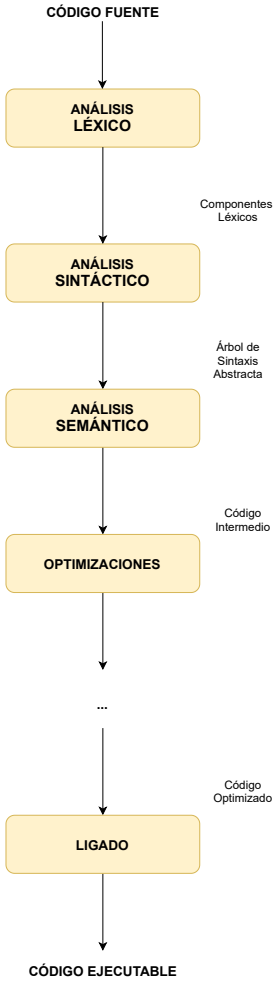

Figura 2.1: Diagrama de bloques de los tipos de análisis para generar código ejecutable.

fases entre la fase de optimización y de ligado, sin embargo para fines de este trabajo de obtención de grado solo se presentan los más representativos, con el objetivo de centrarnos en la fase de análisis semántico.

Cada construcción sintáctica descrita en un código fuente tiene asociada una semántica (ver bloque 3 de la Figura 2.1). Esta última debe ser consistente (congruente) durante su ejecución (ver `CÓDIGO EJECUTABLE` de la Figura 2.1). Sin embargo no toda expresión en tiempo de ejecución mantiene su representación semántica, tal y como se define de manera inicial en el código fuente.

Los intérpretes realizan estos análisis durante tiempo de ejecución: comienzan desde la primer instrucción o expresión escrita en el código fuente, traduciendo cada una de las instrucciones a lenguaje de máquina, ejecutándose en el orden en el que fueron descritas. Por otro lado, el compilador, traduce todo el código fuente a lenguaje de máquina (código ejecutable) en tiempo de compilación. Las instrucciones son almacenadas en el código ejecutable, con el objetivo de que tales instrucciones del código fuente se ejecuten al invocarlas en tiempo de ejecución.Los lenguajes a los que se refiere este trabajo están catalogados como de alto nivel. Éstos traducen el código fuente a lenguaje de máquina (código binario que la computadora puede ejecutar). Para

poder ejecutar estos programas en una computadora, es necesario que un intérprete o compilador convierta los programas escritos en lenguajes de alto nivel (código fuente) a lenguaje de máquina (código binario). Un intérprete traduce cada instrucción en un programa a lenguaje de máquina, ejecutándolas en seguida, es decir, en tiempo de ejecución. Mientras que un compilador traduce todo el código fuente escrito en un lenguaje (de alto nivel) a lenguaje de máquina. Las instrucciones del programa son traducidas y almacenadas en el código ejecutable y cuando éste es invocado, se carga en memoria el programa ya traducido, listo para ejecutarse [7].

Aunque los lenguajes de programación pretenden resolver de manera específica ciertos tipos de problemas en particular, los lenguajes pueden realizar un conjunto de tareas esenciales independientemente de los problemas que resuelvan. Este tipo de categorización de lenguajes es conocido como Lenguajes de Propósito General (*General Purpose Language, GPL*) o bien Lenguajes de Dominio Específico (*Specific Domain Language, SDL*). Por otro lado, un lenguaje de programación se puede clasificar también en términos de sus construcciones [56]:

- Construcciones definidas en el núcleo del lenguaje, dentro de las cuales encontramos a las primitivas predefinidas en el lenguaje.
- Construcciones definidas por el programador, las cuales son hechas dependiendo del tipo de programa que se quiera realizar usando primitivas del lenguaje o bien redefiniendo las ya provistas por el mismo.

Clasificación de semántica

La semántica formal de un lenguaje de programación impacta en la verificación de las construcciones, en tiempo de compilación [15].

Para poder dar un significado a cada uno de los términos de un lenguaje existen tres tipos de enfoques con el fin de formalizar la semántica de un lenguaje según Hoare y Pierce [50, 66]:

La semántica operacional: describe el comportamiento de un lenguaje a través de una máquina abstracta simple para éste. Se dice que la máquina es abstracta porque (a) no se refiere a una computadora específica y (b) porque maneja los términos de un lenguaje como si lo hiciera un procesador, pero sin llegar a manejarlo como un cojunto de instrucciones de bajo nivel.

La semántica denotacional: donde los términos de un lenguaje son tratados como objetos matemáticos y no solo como una máquina de secuencia de estados. En otras palabras, se trata de encontrar una colección de dominios semánticos para posteriormente mapearlos a través de una función de interpretación a elementos contenidos en esos dominios. Esta permite relacionar un programa con su especificación de acuerdo a su comportamiento y propiedades observables.

La semántica axiomática: este enfoque es más directo que los anteriores y en vez de definir el comportamiento de un programa (dada la semántica operacional y denotacional) los métodos axiomáticos toman las *leyes* mismas como la definición del lenguaje.

El enfoque de la semántica operacional es derivado del álgebra. Primero se tiene que establecer una definición algebraica para realizar las relaciones de transición, y cada una de las reglas de transición pueden ser probadas individualmente como teoremas algebraicos, para reducir riesgos complejos o interacciones inesperadas. Por ejemplo, los fenómenos de bloqueo mutuo [2] (sin transiciones) y divergencia (secuencia infinita de transiciones) pueden ser analizados usando el álgebra y así mostrar su interpretación correcta. Este tipo de semántica permite describir las pruebas de manera simple y modular, permitiendo seguir la sucesión natural de las mismas de manera abstracta a una implementación concreta. El objetivo de la semántica operacional es capturar el significado de un programa como una relación en donde se describe cómo se ejecuta un programa. Los otros estilos de semánticas se enfocan más en las estructuras matemáticas para darle significado a un programa (semántica denotacional) o en cómo describir el comportamiento de un programa por medio del razonamiento matemático (semántica axiomática) [50].

La Semántica Operacional Ejecutable, también conocida por sus siglas, SOE, está orientada al significado que se le asocia a cada una de las expresiones en un lenguaje de programación en tiempo de ejecución [85]. Ésta se

[2] Traducción del inglés *deadlock*.

encarga de describir el signficado de un lenguaje por medio de un sistema de transiciones de estado [67]. A su vez, la SOE permite verificar que una expresión (operación definida en el lenguaje) sea consistente y segura en tiempo de ejecución, dicho de otro modo, se ejecute con los tipos de valores esperados definidos por su semántica [86].

Sistema de tipos

La verificación de un programa fundamenta a éste bajo su especificación formal. Por lo anterior, una parte crucial del desarrollo de software es justamente obtener una especificación completa y consistente del programa [16]. En el desarrollo de software se utilizan varios métodos formales para ayudar a verificar los programas en cuanto a su comportamiento. Algunos de estos métodos formales son los marcos de trabajo de la lógica de Hoare [49], la especificación algebraica de lenguajes [50] y la semántica denotacional [66]. Una técnica más que se desarrolla para monitorear en tiempo de ejecución (*run-time monitoring*) el comportamiento de los programas son los sistemas de tipos[3]. Pierce hace mención y uso del concepto de sistema de tipos [4] (o teoría de tipos) para especificar si un programa tipificado estática o dinámicamente presenta un buen comportamiento. Así un sistema de tipos es definido por Pierce como [66]:

> "Un sistema de tipos es un método sintáctico de seguimiento para demostrar la ausencia de ciertos comportamientos del programa, clasificando expresiones según los tipos de valores que éstas calculan" [5].

Un sistema de tipos permite, en cierta forma, tener una aproximación estática al comportamiento de un programa en tiempo de ejecución. Por lo anterior, una manera de categorizar a los lenguajes de programación es por la presencia o ausencia de sus tipos, es decir, por su tipificado que puede ser estático o dinámico. Ambos enfoques presentan ventajas y desventajas tanto para el diseñador del lenguaje como para el programador [90]. De esta forma dependiendo del diseño del lenguaje, el sistema de tipos detecta y previene algunos errores de tipo en tiempo de ejecución [66]. La verificación de tipos estática permite [56, 66]:

- Detectar errores relacionados con las declaraciones de tipos en tiempo de compilación.
- Debido al punto anterior, existen algunos efectos secundarios que pueden ser detectados y tratados más fácilmente por el programador en tiempo de ejecución.
- Los posibles cambios en alguna estructura compleja pueden ser más fácilmente detectados y cambiados por el programador al ser encontrados desde la primer inconsistencia en las declaraciones de los tipos.
- Documentar los programas de tal forma que sean de lectura más sencilla para el programador. La declaración de tipos usadas en las firmas de cada función (método, o procedimiento, entre otros) ayudan a autodocumentar un programa en sí mismo.

Propiedades de un programa

Gregory R. Andrews define los siguientes conceptos en [4] como:

> "Una propiedad de un programa es un atributo que es verdadero en toda posible historia de ese programa, y por lo tanto en todas sus ejecuciones. Cada propiedad puede ser formulada en términos de dos tipos de características especiales: seguridad y vitalidad. La propiedad de *seguridad* se refiere a que un programa nunca entra en un estado equivocado (mal estado), es decir, uno en el que algunas

[3] No todos los componentes de un programa se comportan dinámicamente de acuerdo a lo establecido en su especificación formal [66].

[4] El primer sistema de tipos fue introducido en FORTRAN en el año de 1950, con el fin de calcular expresiones aritméticas diferenciando valores de tipo entero de valores de tipo real [66].

[5] *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifyng phrases according to the kinds of values they compute.* [66].

variables tienen valores no deseados. La propiedad de *vitalidad* afirma que un programa eventualmente entra en un estado bueno, es decir, en el cual las variables tienen todos valores deseables" [6].

La propiedad de **seguridad** se refiere al conjunto de errores en tiempo de ejecución que pueden ser manejados por el sistema de tipos. Un lenguaje seguro según Pierce en [66] es:

"Un lenguaje seguro es aquel que protege sus propias abstracciones" [7].

Los términos lenguaje seguro y verificación de tipos segura tienen un significado distinto. La seguridad en un lenguaje se obtiene por medio de la verificación estática [66].

Krishnamurthi en [56] define la propiedad de **consistencia** (solidez o corrección, del inglés *soundness*) de un sistema de tipos, si:

"El verificador de tipos en tiempo de ejecución puede predecir correctamente los errores de tipo".

Mientras que Pierce lo define en [66] como:

"La seguridad (o solidez) de cada sistema de tipos debe juzgarse con respecto a su propio conjunto de errores en tiempo de ejecución." [8].

Otro término importante de definir es la *seguridad de tipos*, el cual define Shriram Krishnamurthi en [56] como:

"La seguridad de tipos (*type safety*) es la propiedad de que ninguna operación primitiva será aplicada a valores de tipo incorrecto".

Dicho de otro modo, es que toda operación primitiva siempre es aplicada a valores de tipo correcto. Cabe mencionar que esta observación se puede aplicar cada vez que se componen o se usan las operaciones primitivas con otras operaciones permitidas por el lenguaje asegurando que cualquier programa será aplicado a valores adecuados (dados los tipos de dichos valores).

"Al ejecutarse el verificador de tipos sobre un programa, éste debe presentar alguno de los siguientes tres posibles estados para ser así *correcto o consistente* [56]:

- El programa en ejecución termina, regresando e imprimiendo valores del tipo esperado.
- El programa entra en un ciclo infinito (*infinite loop*).
- El programa termina en algún estado previamente bien definido de un conjunto de excepciones predefinidas; es decir, se levanta una excepción de un conjunto de excepciones bien definidas".

Krishnamurthi redefine en [56] el concepto anterior como:

"Para todo programa p, si el tipo de p es T, entonces p será consistente o correcto sí p termina con algún valor v tal que v:T, o bien levanta una excepción de un conjunto de excepciones bien definidas".

Cabe mencionar que éstas son solo algunas de las definiciones asociadas a la seguridad de un lenguaje. Existe toda una discusión aún hoy en día donde la propiedad debería ser aplicada a la implementación del lenguaje y no al lenguaje en sí mismo [99]. Esta discusión está encaminada a identificar el contexto en donde usa la palabra **seguridad**, es decir, seguridad de tipos, seguridad en reclamar, garantizar y proveer espacio en memoria, entre otros. Por ejemplo, cuando se habla de código seguro usando hilos, (traducción del inglés *threads*) se dice que el código es seguro si éste no presenta cierto tipo de errores asociados al uso de *threads* y la memoria compartida,

[6] *A property of a program is an attribute that is true of every possible history of that program, and hence of all executions of the program. Every property can be formulated in terms os two special kinds of propertirs: safety and liveness. A safety property asserts that the program never enters a bad state, i.e., one in which some variables have undesirable values. A liveness property asserts that a program eventually enters a good state, i.e., one in which the variables all have desirable values* [4].

[7] *A safe language is one that protects its own abstractions* [66].

[8] *The safety (or soundness) of each type system must be judged with respect to its own set of run-time errors* [66].

incluyendo la competencia de datos o de hilos y de los bloqueos mutuos. En este trabajo de tesis doctoral nos referiremos a la seguridad asociada a los tipos en un lenguaje de programación.

El adjetivo *estático* algunas veces puede asociarse a la palabra *explícito*. En particular se habla de lenguajes tipificados explícitamente. También lo estático suele considerarse *conservador*. En este sentido algunos autores consideran a los lenguajes estáticos como lenguajes en los cuales no se presentan errores asociados a los tipos en tiempo de ejecución. Dicho de otro modo, tienen un *buen comportamiento*. Lo anterior no necesariamente es cierto, pues dependiendo del lenguaje y haciendo un análisis más profundo de algunos programas, podría no garantizar el buen comportamiento de algunos programas en tiempo de ejecución [66, 93]. Por otro lado, un lenguaje puede no necesariamente tener etiquetas de tipos en su sintaxis para desarrollar un programa, y ser seguro. Un ejemplo de este tipo de lenguajes es Scheme o Racket, que son lenguajes seguros, aún y cuando la verificación de su sistema de tipos no es estática [66].

Por otro lado, los lenguajes no seguros no garantizan que los programas bien tipificados (*well typed*) se comporten de la misma manera en la que están especificados, es decir, que éstos presenten un buen comportamiento en tiempo de ejecución [56, 66]. Por ejemplo, el compilador del lenguaje de programación C no realiza una verificación de su sistema de tipos segura, permitiendo que un programa pueda ejecutarse con algunos errores de tipo o no, ya que este lenguaje puede realizar conversiones implícitas de tipos para no regresar ningún error y ejecutar el programa sin errores [56, 106].

La siguiente tabla 2.1 muestra una posible categorización de algunos lenguajes de programación, dada la clasificación expuesta asociada a la verificación de tipos, es decir, estática o dinámica.

	Verificación de Tipos Estática	**Verificación de Tipos Dinámica**
Seguros	ML, Haskell, Java, entre otros.	LISP, Scheme, Perl, Postscript, entre otros.
No Seguros	C, C++, entre otros.	Ensamblador.

Tabla 2.1: Ejemplos de lenguajes de programación clasificados por la verificación de tipos [56].

Cuando se habla de verificación dinámica o estática no necesariamente implica que en ambos casos las verificaciones de tipos se realicen únicamente en tiempo de ejecución (de forma dinámica) o en tiempo de compilación (de manera estática). Existen algunos lenguajes seguros con verificación de tipos estática que realizan la verificación de límites de un arreglo de manera dinámica. El diseño de sistemas de tipos para lenguajes con tipificado dinámico presenta ciertas características e idioms que hacen realmente complicada la verificación de tipos, por lo que es importante tomar en cuenta tanto el diseño de la sintaxis del lenguaje, como la organización que presenta en un programa [66].

Tipificación en lenguajes de programación

Como ya se ha mencionado, existen varias clasificaciones de lenguajes dependiendo del criterio bajo el cual se quiera catalogar. Una de las clasificaciones más usadas depende de si un lenguaje es compilado o interpretado. En ambos casos los lenguajes reciben un código fuente y regresan un código ejecutable, pasando por los tipos de análisis vistos anteriormente. Sin embargo los lenguajes interpretados realizan tales análisis para generar código ejecutable cada vez que se ejecute el código fuente a través de un intérprete.

Por otro lado, los lenguajes compilados traducen el código fuente a través de un compilador que se encarga de generar el código ejecutable y una vez que se tiene tal código ejecutable, éste es el mismo que se ejecuta en cada ejecución, sin tener que volver a realizar la traducción del código fuente a código ejecutable cada que se quiera ejecutar.

Otra categorización de lenguajes es tomando en cuenta dado el tiempo (compilación o ejecución) en el que se determina su tipificado:

Lenguajes con tipificado estático. Los lenguajes que pertenecen a esta categorización deben haber definido los tipos en cada una de sus expresiones en tiempo de compilación. Toda violación de tipos debe ser identificada antes de que se ejecute un programa, y el código ejecutable no puede ser generado; incluyendo

realizar operaciones de optimización para generar código más eficiente en manejo de espacio, hasta que se verifique la consistencia de tipos de cada una de las expresiones en algún programa [9].

Lenguajes con tipificado dinámico. Dentro de los lenguajes que se encuentran en esta clasificación se verifican los tipos de cada una de las expresiones en un programa en tiempo de ejecución. Algunos ejemplos contenidos en esta clasificación de lenguajes están Ruby, Python y JavaScript [70].

El término **tipificado** está relacionado con el grado de información que se mantiene en un lenguaje al momento de usarse. Los lenguajes entonces pueden clasificarse dado su tipificado en un programa, como lo describe [64]:

Fuertemente tipificados. En este tipo de programas el programador debe especificar el tipo de cada expresión usada en el programa.

Débilmente tipificados. Los programas pueden no tener asociado un tipo a cada una de sus expresiones, es decir, no hay restricciones asociadas al uso que se le da a cada parte escrita de un programa.

Lenguajes con tipificación estática

Los lenguajes con tipificado estático conllevan ciertas ventajas al hacer uso de éstos como programadores, como el ayudar a detectar algunos tipos de errores de manera temprana. La mayoría de estos errores pueden ser detectados en tiempo de compilación. Por ejemplo cuando se quiere sumar un número flotante y un booleano el lenguaje debe protegerse y no realizar dicha operación. Por otro lado la documentación de un programa es más clara cuando se tiene este tipo de lenguajes. Por ejemplo, teniendo la firma de una función en un lenguaje con tipificado estático se está haciendo explícito tanto los tipos de los argumentos que recibe una función como el tipo de regreso de la misma; asimismo el compilador puede realizar optimizaciones sobre el código haciéndolo más eficiente. Por ejemplo, reemplazando llamadas virtuales por llamadas directas cuando se conoce estáticamente el tipo que se recibe. Por último la experiencia del programador al realizar el diseño de un programa se incrementa, por ejemplo cuando se conocen los tipos que reciben las funciones, los Ambientes de Desarrollo Integrados o también conocidos por sus siglas en inglés como IDE [9] pueden presentar un menú de los elementos a los que se les puede aplicar esos argumentos [60].

La verificación de tipos estática es una abstracción del tiempo de compilación con respecto al comportamiento de un programa en tiempo de ejecución, y por lo tanto éste es parcialmente consistente e incompleto [60]. En otras palabras, un programa puede regresar valores incorrectos debido a que las propiedades de éstos valores no pueden ser manejadas por el verificador de tipos, y a su vez pueden existir programas en donde no se detecten ciertos tipos de errores y aún así ejecutarse de manera correcta.

Si la semántica está ligada a una sintaxis definida en cada lenguaje y viceversa, entonces los lenguajes con tipificación explícita o implícita, deben reflejar tal semántica en tiempo de ejecución. Los lenguajes con tipificado dinámico verifican los tipos utilizados en un programa en tiempo de ejecución, mientras que los de tipificado estático los verifican en tiempo de compilación. En ambos casos se tienen ciertas ventajas y desventajas [64, 74].

- Ventajas [44, 56]:

  1. Los errores de tipo son capturados de manera temprana, eliminando así los posibles errores que se pudieran presentar en tiempo de ejecución, facilitando el ciclo de desarrollo de software y garantizando cierta estabilidad cuando se está desarrollando un sistema en la etapa de producción en el área de Ingeniería de Software, dicho de otro modo, se pueden detectar errores de tipo de manera temprana y exhaustiva, explorando todos los posibles estados que podría generar una ejecución de un programa.
  2. El programa puede estar mejor estructurado al conocer los posibles errores de tipo antes de ejecutarlo; es decir, para un programador puede ser más fácil de leer un programa con tipos explícitos y así captura los errores antes de ejecutar el programa.

[9] IDE del inglés *Integrated Development Environment*.

3. Un programa[10] puede llegar a ser de más fácil lectura (dada su su sintaxis con tipos explícita) y si así lo facilita el paradigma bajo el cual está diseñado se podrá reutilizar parte de su código.
4. Un programa con verificación estática permite tener representaciones más eficientes en manejo de espacio, ya que la comprobación de tipos se realiza en tiempo de compilación.
5. La firma de las funciones ayuda a autodocumentar un programa *per se*.
6. Los compiladores pueden optimizar cierta parte del código escrito, usando la información relacionada con los tipos en tiempo de compilación, generando código más eficiente en tiempo de ejecución.
7. Contar con un sistema de tipos ayuda a los programadores a diseñar sus programas.
8. Mejora la modularidad de un programa.

- Desventajas [64]:

1. En un programa no se facilita la reutilización de los componentes de manera sencilla.
2. El código generado es menos reutilizable, resulta ser más prolijo, no es más seguro y el nivel de expresividad del lenguaje es menor que en los lenguajes tipificados dinámicamente.
3. Cada identificador en un programa debe estar declarado con un tipo específico, el cual debe siempre usarse de manera apropiada, es decir, que siempre se use bajo expresiones (operadores comúnmente) que estén previamente definidos para utilizar ese tipo de datos específicos.
4. Dentro del Paradigma de la Orientación a Objetos (P.O.O.) se requiere de tipos específicos para cada tipo de interfaz distinta (si se está haciendo uso de interfaces), lo cual puede generar que se tengan una serie de incompatibilidades entre las interfaces, lo cual puede resultar en una tarea compleja al tratar de identificar los tipos específicos de cada una de las instancias de dichas interfaces, para un programador novato en dicho paradigma de programación.

Lenguajes con tipificación dinámica

Los lenguajes de *scripting* se encuentran dentro la clasificación de lenguajes con tipificado dinámico, como Perl, Python, Rexx, Tcl, Visual Basic y algunos intérpretes de comandos de UNIX. Estos lenguajes utilizan una serie preexistente de componentes útiles de otros lenguajes. Usualmente combinan componentes, por ejemplo algunos *scripts* en UNIX, generarán un resultado dada la concatenación de ciertas instrucciones utilizando tuberías (*pipelines*). Es por lo anterior que estos lenguajes de *scripting* son conocidos como lenguajes de pegado[11]. Con la finalidad de simplificar la tarea de los lenguajes de *scripting*, de conectar componentes, los lenguajes tienden a no tener tipos explícitos (*typeless*). En un lenguaje sin tipos explícitos es mucho más sencillo de unir varios componentes, ya que no existen restricciones a priori en relación a ¿cómo deberían de usarse tales componentes? de tal forma que tanto los componentes como los valores pueden ser representados en una misma forma. Por lo que bajo cualquier situación, un valor o un componente pueden ser utilizados [64].

Los programadores que usan lenguajes tipificados dinámicamente desarrollan rápidamente prototipos de sistemas complejos, por lo que la etapa de mantenimiento resulta difícil. La tipificación implícita es una posible solución a este problema. Estos lenguajes agregan verificación de tipos en tiempo de ejecución, tanto de código tipificado explícitamente como implícitamente, sin perder la corrección y consistencia en el sistema de tipos [87].

En esta categorización de lenguajes, la verificación de tipos se realiza hasta el momento en que un valor sea utilizado. El precio que se paga en cuanto al manejo de eficiencia depende de cuánta información es utilizada. Generalmente los lenguajes de *scripting* son interpretados, mientras que en el desarrollo de sistemas grandes se hace uso de un compilador. El uso de los intérpretes proveee un desarrollo más rápido. A su vez los intérpretes hacen las aplicaciones más flexibles permitiendo a los programadores programar aplicaciones en tiempo de ejecución (generar código *al vuelo.*)

Sin embargo, los lenguajes de *scripting* son menos eficientes que los sistemas generados por un compilador, en cuanto a desempeño. Generalmente en estos lenguajes (de *scripting*) se genera código más pequeño que en los

[10] Inclusive los grandes sistemas [64].
[11] Traducido del término en inglés *gluing*.

programas generados por los lenguajes compilados. Dicho de otra manera, el desempeño de los programas de *scripting* tiende a ser dominado por el desempeño de sus componentes, los cuales generalmente están implementados en lenguajes compilados [64]. Actualmente muchos de los desarrollos en el mundo de la programación usan a los lenguajes de *scripting*, dando como resultado aplicaciones orientadas al *pegado*.

La verificación de tipos en los programas escritos en lenguajes tipificados estáticamente se realiza en tiempo de compilación, por lo que cualquier problema relacionado con los tipos es detectado antes de su ejecución. Lo anterior conlleva el generar código ejecutable optimizado, pues la consistencia de tipos se mantiene en la práctica. Sin embargo, no todos los lenguajes que realizan la verificación de tipos estática pueden garantizar que en tiempo de ejecución no se levante alguna inconsistencia con algún tipo asociado a una expresión, y que éstas a su vez se puedan detectar en tiempo de compilación. Un ejemplo de ello, se presenta cuando algún tipo de información es conocida hasta tiempo de ejecución, como los sistemas distribuidos, que intercambian datos entre diversos programas o procesos, los cuales almacenan algún valor de un tipo arbitrario, y posteriormente éste sea extraído de algún dispositivo de almacenamiento estable con el tipo del valor posiblemente modificado [1, 9].

Un ejemplo de lenguaje que realiza la verificación de tipos de manera estática y no satisface la propiedad de seguridad es C++. Los programas escritos en este lenguaje en tiempo de ejecución interpretan operaciones usando patrones de bits, por lo que al efectuar una operación válida por el hardware entre estos patrones puede dar un resultado válido desde este punto de vista, pero de acuerdo con la semántica, la ejecución continúa. Por otro lado, si la aplicación de la operación es errónea, esto puede desencadenar una interpretación de bits que el hardware rechaza, obteniendo un error de segmentación[12] [93].

Algunas de las ventajas y desventajas que presentan los sistemas con tipificado dinámico son [44, 56]:

- Ventajas:
  1. Tener verificación de tipos dinámica permite ejecutar un programa de forma inmediata.
  2. El nivel de abstracción es alto por lo que los programadores pueden hacer generalizaciones de su código.
  3. Lidiar con el tipo de un valor depende del tiempo de ejecución y esto en ocasiones resulta más sencillo.
  4. Este tipo de lenguajes permiten ejecutar cualquier programa sintácticamente correcto.
  5. Las funciones tipificadas dinámicamente pueden ser reutilizadas en una gran clase de argumentos.
  6. El concepto de tipificación dinámica pareciera más sencillo que el estático ya que un programador solo debe entender el comportamiento de un programa en ejecución, sin necesariamente conocer el comportamiento del verificador del sistema de tipos.
- Desventajas:
  1. No se pueden hacer todas las optimizaciones sobre el código pues de algunas partes se desconoce el tipo de entrada que va a recibir una expresión, hasta tiempo de ejecución.
  2. Algunos de los lenguajes con tipificado dinámico garantizan seguridad en errores de memoria.

Lo descrito anteriormente presenta también algunos inconvenientes o desventajas como en el manejo de polimorfismo. Lenguajes sin anotaciones de tipos permiten al programador omitir tales anotaciones de tipo en las abstracciones lambda. Por ejemplo la función identidad $\lambda x.x$ donde en cada contexto donde sea llamada dicha función deberá instanciarse con el tipo del valor que haya sido pasado como argumento en su aplicación. En este caso, los tipos serán tratados como **variables de tipo** las cuales serán sustituidas o instanciadas con otros tipos. La operación de sustitución de tipos por variables de tipos se puede separar en dos partes: (a) tener una operación de sustitución, llamada sustitución de tipos, σ, que va de variables de tipo a tipos y (b) al aplicar dicha función a un tipo particular T para obtener la instancia σT. Siguiendo el ejemplo anterior, de la abstracción lambda, $\lambda x.x$ con el tipo $X \to X$ se debe definir entonces la operación $\sigma = [X \mapsto Int]$, por lo que si se llamara a dicha abstracción con un valor entero de 4 i.e. $((\lambda x.x)\ 4)$ se puede aplicar σ para el tipo $X \to X$ para obtener $\sigma(X \to X) = Int \to Int$ [66].

[12] En algunos lenguajes de programación se presenta como: *segmentation fault*, o *segfault*.

Aquí surge la pregunta acerca de ¿qué sucede cuando se tiene la misma función (en este caso la función identidad) utilizada en un mismo programa con distintos tipos de instancias? Por ejemplo, usando la sintaxis del lenguaje de programación Racket tenemos el siguiente código [56] donde se utiliza la primitva `let` para realizar la asignación de la función identidad al identificador `id`, el cuerpo del `let` contiene una expresión condicional `if` la cual evaluará primero la aplicación de `(id true)` y ésta a su vez, regresará la constante booleana `true`, por lo que la expresión *then* asociada a la expresión *if*, i.e. `(id 5)` devolverá el valor entero de 5:

```
(let (id (lambda(x) x))
   (if (id true)
       (id 5)
       (id false)))
```

Código 2.1: Ejemplo de uso de *let* en Racket [56].

Una alternativa a esto puede ser el añadir a estas abstracciones lambda sin anotaciones de tipo una regla de correspondencia para la relación con restricciones de tipos. Una solución es hacer varias copias de dicha abstracción lambda e ir eligiendo un nombre de variable de tipo diferente como argumento en cada copia. Otra alternativa, podría ser el considerar una única definición de la abstra cción lambda con una variable de tipo llamemos "invisible", lo que reproducirá que en cada copia de la abstracción se utilice el mismo nombre de variable de tipo. Dependiendo de las construcciones que utilice un lenguaje de programación puede presentarse una característica conocida como **polimorfismo let** (*let-polymorphism*).

El polimorfismo definido en [66] se refiere a: "una variedad de mecanismos del lenguaje que permiten que una sola parte de un programa pueda ser utilizada con diferentes tipos en diferentes contextos".

Un problema relacionado con esto es que si el cuerpo del `let` contiene muchas ocurrencias de la variable de ligado del `let`, entonces el lado derecho de dicho `let` (donde se especifica el cuerpo la expresión asociada al nombre del identificador) deberá verificar que no ocurra ese mismo nombre de variable en ésta, ya sea que se tengan o no abstracciones lambda con anotaciones de tipo implícitamente. Y el mismo lado derecho del `let` puede contener otras variables ligadas. Lo anterior repercutirá en una cantidad de trabajo exponencial, dado el tamaño de la expresión original [66].

Para dar una posible solución al panorama descrito en el párrafo anterior se ha propuesto añadir un tipo *dinámico* (del inglés *dynamic type*), el cual está formado por un valor `v` y una etiqueta de tipo `T`, tales que `v` tiene asociado el tipo `T`. Un ejemplo de instancia de un tipo *dinámico* implementado en los lenguajes ha sido el constructor `typecase` [1]. Algunas expresiones en estos lenguajes, pueden entonces inferir los tipos de algunas expresiones en tiempo de compilación y otras hasta tiempo de ejecución, pues hasta ese momento se conocerán los tipos de éstas [9].

Cabe destacar el caso del polimorfismo *ad-hoc*, en el cual es permitido que un valor polimórfico presente diferentes comportamientos cuando *"ve"* o identifica diferentes tipos. Uno de los usos más comúnes de este tipo de polimorfismo es la sobrecarga de operadores, la cual asocia un mismo identificador o nombre de función a muchas implementaciones. Así, el compilador, dependiendo de si realiza la resolución de la sobrecarga de manera estática o dinámica, elige la implementación adecuada para cada aplicación de función, dados los tipos de sus argumentos. Una estructura provista por algunos lenguajes de programación para usar este tipo de polimorfismo es através de la primitiva `typecase`, la cual permite la coincidencia de patrones [13] dados los tipos en tiempo de ejecución [66].

Durante la última década, considerables lenguajes de programación de *scripting* han sido usados tanto en la industria como en proyectos de código abierto[14]. Éstos han ido suplantando a los lenguajes dominantes (C++ y Java). Por ejemplo, en el área de administración de sistemas y programación Web. Los lenguajes de *scripting* con tipificación dinámica presentan en principio una ventaja en cuanto al grado de libertad que los programadores de estos lenguajes pueden tener; sin embargo, esta libertad trae consigo una desventaja ya que los lenguajes de esta categoría al no contar con un sistema de tipos, carecen de herramientas (de mantenimiento) de sistemas suficientes para realizar dicha tarea, provocando cierto tipo de errores en tiempo de ejecución [34].

La solución para este tipo de lenguajes (con tipificado dinámico) ha sido poder tipificarlos de algún modo, lo cual conlleva algunas desventajas como (a) modificar el código existente para poder introducir tipos, (b) dado el

[13] Traducción del inglés *pattern matching*.
[14] Traducción del inglés *open-source*.

punto anterior, el código nuevo (con tipos) puede contener errores, sin mencionar el esfuerzo que representa para los programadores de este tipo de lenguajes al tener que migrar sus programas actuales sin tipos a versiones con tipos, y (c) los lenguajes existentes convencionales usados en la industria no parecen ser compatibles con los lenguajes con tipificado dinámico, como son los lenguajes de *scripting*. Por otro lado, el poder tipificar los lenguajes dinámicos (a) ayuda a preservar una interoperabilidad consistente entre los dos mundos en lenguajes (con tipos y sin tipos), (b) a su vez los *idioms* del lenguaje (con tipificado dinámico) se mantienen tanto como sea posible, para así enriquecer tanto los módulos con tipos como los que no tienen tipos explícitos, dando como resultado un código consistente con pequeños cambios [34].

El mejor de los mundos es uno donde el sistema de tipos sea flexible y pueda verificar cualquier programa tipificado dinámicamente de la siguiente forma [44]:

- Los sistemas de tipos estándar pueden probar que un programa es correcto [15]. Donde un conjunto de programas correctos son casi siempre recursivos numerables, por lo que el verificador de tipos decidible debe rechazar algunos programas correctos.
- Los programas con tipificado dinámico se basan fuertemente en la unión de tipos recursivos, y estructuras con subtipificado. Muchas de estas características hacen la inferencia y verificación de tipos más difícil.

La filosofía detrás de los lenguajes con tipificado dinámico se fundamenta en sus datos y las operaciones que se realizan sobre éstos. Por otro lado las funciones basan sus reglas y estructura en el algortimo de subtipificado, guiado por sus demostraciones. Varios enfoques en el área han tratado de resolver este tipo de problemas (del tipificado dinámico). Según Ben Greenman existen dos enfoques principales:

1. Ayudar al lenguaje por medio del suficiente número de *idioms*.
2. Promover un sistema de tipos suave para que el verificador de tipos pueda inferir los tipos (o dar información) de un programa de manera semántica, tal que garantice que ese programa es seguro.

Así el diseño un sistema verificador de tipos suaves debe de satisfacer los siguientes puntos [44]:

- Todo programa sintácticamente incorrecto debe ser excluído (es decir, el sistema verificador de tipos debe rechazarlo).
- Todo programa debe ejecutarse de forma segura.
- El proceso de verificación de tipos debe ser discreto, donde discreto está caracterizado por dos principios:
  - Principio del texto mínimo (*minimal text principle*): el sistema de verificación de tipos debe de funcionar aún y cuando el programador no haya escrito los tipos en un programa.
  - Principio de la falla mínima (*minimal failure principle*): el sistema verificador de tipos debe ejecutar alguna función para garantizar que no se produzcan errores asociados a los tipos en tiempo de ejecución.

Por otro lado, los lenguajes de programación que presentan tipificación dinámica, donde los tipos asociados a cada una de las construcciones en un código fuente se determinan en tiempo de ejecución, se han estudiado para establecer las relaciones que deben de mantener cada una de las expresiones definidas por el programador y el código generado. Así, las instrucciones en un programa deben de preservar sus propiedades (por ejemplo en la consistencia de sus tipos) tanto en su especificación formal como en tiempo de ejecución, de manera independiente [66].

Cada vez que un valor (con su semántica) pasa de programa no tipificado a uno tipificado en tiempo de ejecución se debe asegurar que el valor se mantenga con el tipo especificado. Un valor plano (*flat value*), digamos un número, requiere el mismo tipo de comprobación que cualquier otro valor en un lenguaje no tipificado. Cuando un valor de orden superior, como una función o un objeto, cruza ciertos límites, la verificación de una

[15] Traducción del término en inglés *good*. En este contexto se interpreta este adjetivo como un programa que no genera un error de segmentación.

propiedad de tipo es imposible, porque semánticamente, estos valores son infinitos. Por lo tanto, los sistemas de verificación de tipos, retrasan las verificaciones pertinentes hasta tiempo de ejecución, es decir, cuando se aplique una función, se envíe un mensaje a un objeto, entre otros. Por otro lado, para valores compuestos, como un arreglo, la verificación en tiempo de ejecución inspecciona todos los elementos aunque ninguno se acceda realmente, o bien retrasa los chequeos hasta que se ejecuta un acceso. Cabe mencionar que la siguiente discusión está basado en lo expuesto por Tibon et al. en [93].

Todo tipo de verificación trae consigo algún costo. Mientras que la verificación de valores planos o simples es relativamente barata, el costo de otras construcciones puede ser no-trivial. El lenguaje Typed Racket presenta dos tipos de costos: (a) la asignación de envolturas (*wrappers*) para retrasar la verificación y (b) el tiempo para realizar la verificación ejecución.

Para ejemplificar en lo anterior, supongamos que se tiene un programa donde una función tipificada fluye a una parte del código que no está tipificada. Si el código que no está tipificado aplica la función un millón de veces, el argumento se verifica también ese mismo número de veces. Si además, la función ya envuelta vuelve a salir del ambiente tipificado en un código no tipificado, la verificación se vuelve a hacer tanto en la envoltura, como en cada aplicación.

Para evitar el costo de cruzar los ambientes, se migra Racket en módulos que contienen algunas funciones, las cuales son manejadas como unidades. Algunas de estas funciones son visibles a los módulos clientes y otras se mantienen escondidas. En pocas palabras, se mantienen pocos cruces en estos módulos (es decir, módulos pequeños) para así reducir el costo de la verificación en tiempo de ejecución.

El desempeño ha sido uno de los más grandes problemas a resolver durante los últimos años en el área, justificando así una de las mayores preocupaciones acerca del costo en tiempo de ejecución en los lenguajes con verificación dinámica.

Las evaluaciones de desempeño sugieren tres cuestiones a considerar [93]:

1. La consistencia o corrección (*soundness*) en un programa se desea obtener, pero no es eficiente. Por un lado los investigadores quieren convencer a la industria que la consistencia en un sistema de tipos es una idea factible. Solo se deben desarrollar métodos de evaluación para llegar a ello y prestar atención en cada paso del desempeño.

2. Las implementaciones de retroadaptación son difíciles. Buscar estrategias de implementación para optimizar el compilador justo a tiempo con el fin de obtener alguna ventaja de la verificación en tiempo de ejecución.

Poder combinar tipificado dinámico y estático en un mismo lenguaje conlleva ciertas ventajas para los programadores. Por un lado, el tipificado dinámico se requiere cuando se desarrollan sistemas rápidos de prototipos, y se utilizan estructuras de datos heterogéneas. Por otro lado, el tipificado estático proporciona seguridad, consistencia, corrección y eficiencia en la programación [104].

Interpretación abstracta

La interpretación abstracta es definida por Cousot en [29] como:

> "Una teoría general para aproximar la semántica de sistemas dinámicos discretos".

La interpretación abstracta se usa principalmente como una guía para evitar errores conceptuales haciendo uso de una metodología para diseñar especificaciones formales. Es utilizada en temas relacionados como los sistemas de tipos, cálculo lógicos, semántica y verificación de programas, análisis de programas, evaluación parcial, y semántica de cálculos abstracta, generación de pruebas para depurar un programa, inferencia de tipos polimórfica, verificación de sistemas híbridos, entre otros. La idea general de analizar un programa se puede puede expresar en la relación costo-beneficio que se obtiene al programar. Por ejemplo, ¿qué es más valioso?, ¿invertir ocho horas por la noche procesando u ocho horas al día, tratando de encontrar un error en la programación?

Algunas de las aplicaciones de la interpretación abstracta son para la especificación de analizadores de programas cuando se quiere ver el desempeño de los compiladores, en particular cuando se requiere generar código optimizado. Regularmente el desarrollo de software es un tarea difícil, depende del número de programadores,

el alcance del proyecto, el tiempo estimado para cada actividad, etcétera. Es por ello que el generar código optimizado resulta ser una tarea imprescindible en este proceso. En particular se dice que el código generado después de la etapa de optimización de código no presenta el mismo comportamiento que el código escrito en el código fuente. El problema es que el código generado después de las optimizaciones suele no coincidir con el código escrito inicialmente, dando como resultado inconsistencias en la evaluación. Dicho de otro modo, la etapa de optimización ha afectado la semántica del programa. La interpretación abstracta es usada en algunos tipos de análisis para garantizar algunas propiedades en un programa como la corrección y seguridad en tiempo de ejecución, como en los siguientes ejemplos [2]:

- Análisis riguroso: este tipo de análisis realiza una optimización sobre lenguajes funcionales perezosos, identificando los argumentos que se pasan por valor (para no crear *closures* si no es necesario).
- Análisis de actualización: este análisis permite determinar en qué momento es seguro realizar la eliminación de un objeto (cuando ya no existe ninguna referencia a éste).
- Análisis de cláusulas significativas: este tipo de análisis ayuda a identificar las partes de código de una función que son relevantes (significativas) en una aplicación específica y así reducir una sobre carga.
- Análisis de modo: este tipo de análisis se presenta en los intérpretes de Prolog cuando se detecta si una variable lógica se utiliza exclusivamente como una variable de entrada o salida.

La notación formal utilizada para definir y usar la interpretación abstracta es propuesta por Cousot [29] como sigue[16]:

La semántica de un lenguaje L es denotada como S_{sd} y la semántica estándar se representa como $S_{sd}[\![P]\!] \in \mathcal{B}$ de cada programa P en el lenguaje, donde B_{sd} se refiere al comportamiento estándar de un programa.

La semántica del dominio puede ser usada para un sistema de transiciones, funciones de orden superior, entre otros.

Existen varios enfoques para analizar un programa. Los autores Flemming Nielson, Hanne Nielson y Cris Hankin proponen los siguientes: (1) análisis de flujo de datos, (2) análisis basado en restricciones, (3) interpretación abstracta y (4) sistemas de tipos y efecto.

Analizar programas de manera estática en tiempo de compilación permite predecir ciertas aproximaciones acerca del comportamiento (valores) que tiene este programa en tiempo de ejecución o de manera dinámica. Una de las principales aplicaciones de esto conlleva el generar código eficiente, es decir, evitando código redundante por ejemplo al realizar ciertos cálculos, reducir el comportamiento involuntario o malicioso de algún programa. Este tipo de enfoques permite analizar un programa para proveer aproximaciones de posibles respuestas (*approximate answers*) correctas tanto estática como dinámicamente. Para ello se realizan análisis de programas basados en su semántica, es decir, la información obtenida del análisis permite probar si un programa es seguro (o correcto) con respecto a la semántica del lenguaje. Nielson et al. presentan varios enfoques que permiten analizar de manera formal un programa para verificar si es correcto o no respecto a la semántica del lenguaje. Los autores consideran un lenguaje imperativo llamado WHILE el cual está formado por expresiones aritméticas, booleanas y enunciados que operan sobre variables, números y etiquetas de la siguiente forma [63]:

Expresiones aritméticas: $a \in AExp$

Expresiones booleanas: $b \in BExp$

Enunciados: $S \in Stmt$

Variables: $x, y \in Var$

Números: $n \in Num$

Etiquetas: $\ell \in Lab$

[16] Las ideas preliminares de Cousot se inspiran por el trabajo de Mycroft, el cual se basa en un análisis de diagramas de flujo, donde un programa se representa por medio de gráficas, donde los nodos se refieren a las operaciones de un programa y las aristas representan el control de flujo [2].

Operadores aritméticos: $op_a \in OP_a$

Operadores booleanos: $op_b \in OP_b$

Operadores relacionales: $op_r \in OP_r$

La sintaxis abstracta entonces está dada por:

$$a ::= x \mid n \mid a_1\, op_a\, a_2$$
$$b ::= true \mid false \mid not\, b \mid b_1\, op_b\, b_2 \mid a_1\, op_r\, a_2$$
$$S ::= [x := a]^\ell \mid S_1; S_2 \mid if\, [b]^\ell\, then\, S_1\, else\, S_2 \mid while\, [b]^l\, do\, S$$

Para realizar el análisis de un programa haciendo uso de la interpretación abstracta se deben de identificar y nombrar (etiquetar) a cada una de las subexpresiones de la expresión a analizar (programa), y se realiza el análisis de definiciones de alcance (*Reaching Definitions Analysis, RDA*). Por ejemplo, en el programa que calcula el factorial de un número etiquetamos seis expresiones:

$$[y := x]^1; [z := 1]^2; while\, [y > 1]^3\, do\, ([z := z * y]^4; [y := y + 1]^5;); [y := 0]^6;$$

Lo anterior se puede leer como una asignación de la forma $[x := a]^\ell$ de un programa, se define como que a x se le asigna un valor establecido en la expresión aritmética a dada la etiqueta ℓ.

De los cuatro tipos de análisis que se presentan en el libro, se presenta la **interpretación abstracta**, la cual está definida por Nielsen et al. como [63]:

> "Una metodología general para realizar un análisis de un programa el cual es independiente de su estilo de especificación."

Para llevar a cabo este análisis se realiza una recolección semántica[17] la cual registra o mantiene un conjunto de rutas, denotadas por `tr` [18] desde las cuales se puede alcanzar un punto en un programa:

$$tr \in Trace = (Var \times Lab)^\star$$

Las rutas se almacenan con los valores que van obteniendo las variables en el transcurso de un cálculo. Un ejemplo de uso de la interpretación abstracta en el programa que calcula el factorial de un número suponiendo que el cuerpo del `while` se ejecuta dos veces, se ve así:

$$((x,?),(y,?),(z,?),(y,1),(z,2),(z,4),(y,5),(z,4),(y,5),(y,6))$$

Las rutas entonces deben de contener la información suficiente para obtener el conjunto de Definiciones de Alcance Semánticamente (del inglés *Semantically Reaching Definitions, SRD*).:

$$SRD(tr)(x) = \ell \text{ iff el par más a la derecha } (x, \ell') \text{ en } tr \text{ contiene } \ell = \ell'$$

De tal forma que el dominio `(DOM)` de la ruta descrito como `DOM(tr)` para un conjunto de variables se define la `SDR(tr)`, es decir, $x \in DOM(tr)$ si y solo si para algún par (x, ℓ) ocurre en tr.

La recopilación semántica (*collecting semantics, CS*) especifica un superconjunto de posibles rutas en varios puntos de un programa. Así, el alcance de las definiciones (*reaching definitions, RD*) representa un conjunto de definiciones que pueden alcanzarse desde la entrada a la etiqueta ℓ, denotado por la siguiente nomenclatura: $RD_{entry}(\ell)$.

$$\forall x \in DOM(tr) : (x, SRD(tr)(x)) \in RD_{entry}(\ell)$$

Por otro lado, el trabajo de Nielsen et al. establece que las relaciones entre las rutas están dadas por las **conexiones de Galois**, utilizando (1) una función de abstracción α y (2) una función de concretización γ.

[17] Recolección semántica se tradujo del inglés *collecting semantics*.

[18] Del inglés *traces*, también puede asociarse a los términos en español trayectoria o traza.

El tipo de análisis que se realiza con base en dos operadores (funciones) las cuales se definen:

$$\vec{\alpha}(X_1, \ldots, X_n) = (\alpha(X_1), \ldots, \alpha(X_n))$$
$$\vec{\gamma}(Y_1, \ldots, Y_n) = (\gamma(Y_1), \ldots, \gamma(Y_n))$$

La semántica de un lenguaje de programación identifica un conjunto de valores `V` (como estados, cerraduras, reales de doble precisión, entre otros), y a su vez especifica cómo se transforma un valor v_1 en otro valor v_2 en un programa p. Lo anterior se escribe de la siguiente forma:

$$p \vdash v_1 \rightsquigarrow v_2$$

Asimismo se identifican ciertas propiedades, denotadas por L (como cerraduras abstractas, cotas superiores e inferiores para números reales, etcétera). Estas propiedades deben expresar cómo una propiedad l_1 se convierte en otra propiedad l_2 en un programa p. Este tipo de comportamiento se expresa de la siguiente forma:

$$p \vdash l_1 \triangleright l_2$$

El símbolo $\triangleright$ es requerido para definir una función determinística de la forma $f_p(l_1) = l_2$ lo cual tiene el siguiente significado $p \vdash l_1 \triangleright l_2$.

Este enfoque únicamente se aplica para analizar propiedades de un programa que describen un conjunto de valores, de tal forma que cada análisis de un programa debe ser correcto con respecto a su semántica. Por ejemplo, para analizar un tipo de programas, su semántica está directamente relacionada con las propiedades de sus valores usando la siguiente relación de corrección (*correctness relation*):

$$R : V \times L \to \{true, false\}$$

La relación de corrección `R` de un programa es útil para garantizar que tal relación se va a preservar durante ejecución. Así si la relación se mantiene entre el valor y la propiedad inicial, entonces éstas también se preservan entre el valor y la propiedad final:

$$v_1 \, R \, l_1 \wedge p \vdash v1 \rightsquigarrow v_2 \wedge p \vdash l_1 \triangleright l_2 \Rightarrow v_2 \, R \, l_2$$

Se dice entonces que una relación `R` que satisfaga este tipo de condición es una *relación lógica*. En el ámbito de la Interpretación Abstracta se considera un conjunto de propiedades `L` bajo una estructura en preorden y una relación `R` de corrección.

Por otro lado, se puede usar un enfoque alternativo al usar la relación de corrección entre valores y propiedades $R : V \times L \to \{true, false\}$ utilizando una función de representación:

$$\beta : V \to L$$

La idea es que a través de la función β se mapee un valor a la mejor propiedad que lo describa. El criterio de corrección para su análisis debe seguir:

$$\beta(v_1) \sqsubseteq l_1 \wedge p \vdash v_1 \rightsquigarrow v_2 \wedge p \vdash l_1 \triangleright l_2 \Rightarrow \beta(v_2) \sqsubseteq l_2$$

La idea es que si el valor inicial v_1 está descrito de forma segura por la propiedad l_1, entonces el valor final v_2 se describe de manera segura por la propiedad l_2.

Una breve introducción al tipificado gradual

Se han desarrollado algunos modelos para ayudar a conceptualizar, estructurar y evaluar los diferentes diseños de lenguajes tanto de manera formal como en la práctica. Por mencionar algunos de esos enfoques se tiene el *Tipificado Gradual (TG)* [81]; el *Tipificado Híbrido (TH)* [60], el *Tipificado Suave (TS)* [93] [19], entre otros. Estas

[19] Tipificado suave, se tradujo del concepto *soft typing* en inglés.

son algunas de las propuestas formales estudiadas en el área de lenguajes de programación, para especificar la semántica de tipos en términos de los tipos estáticos ya existentes en un lenguaje [40].

Los autores Siek y Taha [78] son los primeros investigadores en incluir el término Tipificado Gradual (TG) y con éste, la relación entre la semántica estática y dinámica de un lenguaje. A partir de este punto, se han hecho estudios formales de algunos lenguajes con Tipificado Gradual y lenguajes con tipificado dinámico, con el objetivo de formalizar la relación de tales lenguajes tanto en la parte estática como en tiempo de ejecución.

Algunos investigadores del área de lenguajes de programación han estudiado diversos enfoques relacionados con el Tipificado Gradual en varios lenguajes de distintos paradigmas como el Orientado a Objetos (O.O.) y funcional; y a su vez cubriendo varios planteamientos como el estado de tipos (*typestates*), propiedades de los tipos, tipificado seguro, así como sus efectos, entre otros. Sin embargo, aún hay varias interrogantes y estudios profundos en algunos lenguajes de programación para los cuales no se han podido realizar el diseño de la parte formal de las construcciones ya implementadas, sobre todo en lenguajes con Tipificado Gradual [40].

En particular, los programas escritos en lenguajes con tipificación gradual o híbrida pueden ser inconsistentes ya que garantizar que se evalúen siempre bajo el mismo sistema de tipos estático en tiempo de ejecución, no necesariamente da el resultado esperado, consistente y correcto dadas las reglas semánticas de tipos. Las construcciones que se definen en un código fuente deben ser consistentes tanto en la especificación formal como en tiempo de ejecución [78]. Para que esto sea posible, en este trabajo se provee un enfoque basado en la Tipificación Gradual, el cual satisfaga la noción de consistencia de tipos utilizada en el tipificado gradual expuesto por Garcia et al. en [42], la cual se basa en un sistema de tipos estático pre-existentes.

Metodología de Tipificado Gradual usando Tipos Unión

Una teoría científica se compone de descripciones que identifican cierta clase de procesos del mundo real. De esta forma tanto las propiedades observables como el comportamiento de tales procesos pueden ser formalizados a partir de una teoría matemática, usando la deducción o algún tipo de cálculo matemático. Las especificaciones pueden describir una variedad de propiedades observables y comportamiento de algún sistema, incluso si éste no existe aún en el mundo real [50].

Toro y Tanter en [95] muestran un nuevo enfoque que combina la interpretación abstracta para desarrollar tipificado gradual y la unión de tipos. Por un lado, el Tipificado Gradual es una metodología que tiene como objetivo conjuntar las ideologías de los lenguajes con tipificado estático y dinámico en un mismo lenguaje. Por ejemplo, el tipificado gradual permite a los programadores escribir programas con tipificado dinámico en una etapa inicial (prototipos de desarrollo de software) e ir modificando y ampliando estos programas (o sistemas) en piezas de código con tipificado estático, para así tener como resultado programas que no presenten errores en tiempo de ejecución [51]. El Tipificado Gradual a su vez ayuda a los programadores a convertir sus códigos no tipificados explícitamente en programas tipificados estáticamente, obteniendo los beneficios de las anotaciones de tipos explícitos en desempeño y en la detección de errores de tipo de manera temprana [69]. Sin embargo, este tipo de lenguajes presentan algunos problemas al permitir omitir ciertas anotaciones de tipos. Lo anterior provoca la introducción de los tipos dinámicos o desconocidos (*dynamic* y *unknown* respectivamente). Bajo este modelo se pretende mezclar tanto las ventajas de la unión de tipos como del Tipificado Gradual. La metodología utilizada para derivar la semántica estática y dinámica de un lenguaje es provista por medio del Tipificado Gradual Abstracto, TGA [95].

Con el transcurso de los años, el enfoque del Tipificado Gradual se ha estado aplicando a más lenguajes de programación, soportando nuevas características en éstos como son los objetos, el polimorfismo, y la inferencia de tipos. Trabajos recientes muestran de manera formal cómo se puede aplicar este modelo de tipificado haciendo uso de la interpretación abstracta.

El TGA ayuda a reforzar la corrección de tipos dentro del tipificado gradual para tratar con las **imprecisiones** asociadas a los tipos que se puedan presentar en un programa de algún lenguaje en esta categoría de tipificado. Por un lado, los lenguajes tipificados dinámicamente muestran una gran cantidad de información estática poco precisa relacionada con los tipos. Por ejemplo, si se tiene una función f dentro de un lenguaje con tipificado gradual, tal que el tipo que recibe es un entero y se desconoce el tipo de su regreso $Int \rightarrow ?$ el lenguaje con tipificado gradual podrá determinar el tipo que devuelve de manera estática, ya que su evaluación depende del

argumento al que se asocie el parámentro de entrada. De esta forma, si se manda llamar a la función con el valor de tipo entero 1 entonces al ejecutar por ejemplo $((f\ 1)\ +\ 2)$, la verificación de tipos estática se realiza de manera exitosa pues el argumento que reciba la función será de tipo entero y posteriormente el resultado de haber hecho esa primera aplicación se podrá sumar con el valor 2, dando como resultado un valor de tipo entero (Int), es decir, la suma para poder evaluarse de manera adecuada deberá recibir argumentos de tipos entero, por lo que podríamos esperar que la aplicación de función $(f\ 1)$ regrese un valor de tipo entero. Dada la relación de consistencia de tipos se deberá esperar que el tipo del primer argumento sea de tipo entero, con lo cual se "forza" el resultado de $(f\ 1)$ a que sea entero, por lo que el tipo ? definido para la función f como $Int \rightarrow ?$ deberá ser Int para poder ser evaluado en la suma, es decir $? + Int \rightarrow Int$. Por otro lado, si la función f es llamada con el valor booleano *true* el programa ni siquiera podrá realizar la operación asociada al cuerpo de la función, pues la suma no puede ser aplicada a un valor de tipo booleano.

Otro ejemplo donde se puede observar la precisión de tipos es el siguiente. Supongamos que tenemos una aplicación de función cuyo primer argumento es la función que espera recibir un valor de tipo entero o un valor booleano i.e. $Int \vee Bool$ y regresa un valor de tipo desconocido (?), así la función tendría disponible asociados los siguientes tipos $Int \vee Bool \rightarrow ?$ y como argumento de la aplicación se tiene un entero por ejemplo el valor booleano $false$. La aplicación de función completa se puede ve así $((\lambda x : Int \vee Bool.if\ x\ then\ 2\ else\ x+1)\ false)$. Aquí el argumento de la función x es inicializado por el valor booleano $false$ lo cual al evaluarse en el cuerpo de la función dentro de la expresión condicional del `if` regresará el valor de falso lo cual deberá generar la evaluación de la rama de la expresión `else` que es $x+1$ la cual a su vez regresará un error asociado a la operación suma pues no podrá operar con un valor booleano con un entero, pues la suma no está definida para valores booleanos.

Por otro lado, cualquier lenguaje que permita la operación de unión de tipos facilita que un término pueda estar relacionado con varios tipos posibles. Los enfoques clásicos en la unión de tipos presentan uniones de tipos con y sin etiquetas. Centrándose en el problema asociado a la precisión de tipos, se han estudiado dos enfoques para tratar de solventar este problema: (a) unión de tipos disjuntos o etiquetado de tipos, en el cual se trabaja con la suma de tipos $T_1 + T_2$ y las variantes de tipo, y (b) la unión de tipos sin etiquetar, denotados frecuentemente por $T_1 \vee T_2$.

A continuación se describen las uniones etiquetadas y sin etiquetar [66]:

- Uniones etiquetadas, sumas, variantes de tipos o uniones de tipos disjuntos: éstas contienen a los valores de posibles tipos diferentes. Es decir, si los elementos son explícitamente etiquetados entonces éstos pertenecen a un tipo específico, por lo que éste es diferente de cualquier otro. Este tipo de uniones preserva la propiedad de seguridad y no ambigüedad de tipos [20].

  El tipo $T_1 + T_2$ representa la unión de T_1 y T_2 en el sentido de que están incluidos todos los elementos de T_1 y de T_2. Esta unión es disjunta porque los conjuntos de elementos de T_1 o T_2 se etiquetan con *inl*, del inglés *in left* o *inr*, del inglés *in right*, respectivamente, antes de combinarlos, de tal forma que siempre se conoce si un elemento en la unión proviene de T_1 (del lado izquierdo) o de T_2 (del lado derecho).

  Un ejemplo de este tipo de uniones etiquetadas es la suma de dos elementos, denotada como $T_1 + T_2$ donde tanto del lado izquierdo de una expresión *inl*, como del lado derecho *inr* espera recibir un número entero (por ejemplo), y regresar un número entero. Sin embargo, si se recibe del lado izquierdo un entero, 10 y del lado derecho un booleano, de la forma $10 :: Int + Bool$ se puede concluir que tal programa está mal tipificado. Para eliminar esa ambigüedad, se puede hacer un análisis de casos explícitamente, considerando cada etiqueta del programa. Por ejemplo, $\lambda x : Int\ +\ Bool.case\ x\ of\ inl\ x \implies x+1 \mid inr\ x \implies if\ x\ then\ 1\ else\ 0$.

- Uniones sin etiquetar: también se conocen como unión de tipos sin etiquetar o no disjunta. Se representan por la unión de valores de tipo T_1 y T_2. La notación formal para denotar este tipo de uniones es $T_1 \vee T_2$.

[20] Aún en lenguajes con tipificado estático, se pueden presentar algunas situaciones donde existan datos que no puedan ser determinados en tiempos de compilación. Estas situaciones ocurren cuando los datos se encuentran distribuidos en múltiples computadoras, por ejemplo cuando los datos son almacenados en bases de datos o sistemas de archivos externos. Para manejar este tipo de situaciones de manera segura, muchos lenguajes permiten revisar los tipos de los valores en tiempo de ejecución. Una manera de lidiar con esto es usando el tipo *Dynamic* cuyos valores son pares de valores v y un tipo T, donde v tiene asociado el tipo T. Algunas instancias de este tipo dinámico (*Dynamic*) son construidos por medio de un constructor de etiquetado explícito y son inspeccionados con una construcción de tipo segura *typecase*. En ese sentido, el tipo *Dynamic* puede ser pensado como la unión disjunta infinita, cuyas etiquetas son los tipos [66].

Este tipo de unión denota la unión comun del conjunto de valores pertenecientes a T_1 y el conjunto de valores pertenecientes a T_2, sin agregar algún tipo de etiqueta para identificar el origen de un elemento específico.

Un ejemplo de este tipo de uniones sin etiquetar permite que en las ramas `then` y `else` de una condicional `if` se utilicen tipos distintos y éstos a su vez no presenten ningún tipo de relación entre ellos. Por ejemplo, la función $\lambda x : Bool.if\ x\ then\ 1\ else\ false$ se encuentra bien tipificada: $Bool \to Int \vee Bool$. Esto permite que el verificador de tipos acepte programas que presentan ambigüedades o imprecisiones aún usando la unión de tipos.

La unión de tipos sin etiquetar tampoco presenta constructores de proyección, las únicas operaciones seguras para valores de tipo $T_1 \vee T_2$ son las que están soportadas tanto para T_1 como para T_2.

La principal diferencia entre tipos de uniones disjuntas y no disjuntas es que en estos últimos casos carecen de cualquier tipo de construcción de casos. Este caso si un valor v tiene asociado el tipo $T_1 \vee T_2$, entonces la seguridad de las operaciones permitidas sobre v serán solamente las que tengan sentido tanto para T_1 como para T_2.

Dada la problemática anterior, el único enfoque seguro para manejar estáticamente este tipo de imprecisiones es la unión de ambos tipos: con etiquetado y sin etiquetado, usando un tipo gradual $T_1 \oplus T_2$ el cual representa los tipos T_1 y T_2.

Por ejemplo, si se tiene la función $\lambda x : Int \oplus Bool.\, x + 1$ la cual dada la TG se encuentra bien tipificada, ya que x puede ser un entero o un booleano sin utilizar ningún tipo de proyección explícita o análisis de casos. Al momento de aplicar a la función el valor de 1 y el valor de *true*, es decir, $(f\ 1)$ y $(f\ true)$ respectivamente, se concluye que la expresión se encuentra bien tipificada y al hacer la inyección a la TG se preserva la unión implícita de tipos. La expresión en el primer caso evalúa el cuerpo de la función al valor entero 2 y el programa se ejecutará correctamente; sin embargo, en el segundo caso cuando se le pasa un booleano el programa producirá un error en tiempo de ejecución ya que no se pudo hacer una conversión de tipo implícita de manera correcta.

La siguiente tabla compara las uniones de tipos [95]:

	Inyección	**Proyección**	**Uso**
Uniones etiquetadas			
Suma	Explícita	Explícita	Completo
Pruebas de tipos/Conversión de tipos	Implícita	Explícita	
Uniones sin etiquetar	Ninguna	Ninguna	Restringido
Uniones graduales	Implícita	Implícita	Completo

Tabla 2.2: Operador de conjunción sobre valores del resultado del algoritmo.

Las conversiones de tipo (*cast* en inglés) pueden clasificarse en *casting* hacia arriba (*up cast*) y *casting* hacia abajo (*down cast*), en éstas se realiza la acción de atribuir un tipo T a una expresión e explícitamente. La conversión de tipos hacia arriba (*up cast*) se presenta cuando a un término se le asigna un super tipo del tipo esperado. Esto puede verse como una forma de *abstracción*, una forma de esconder algunas partes de un valor. Por ejemplo, si tenemos un término que es un registro podemos esconder algunos de sus campos, usando el *casting* hacia arriba. Por otro lado, el *casting* hacia abajo asigna a una expresión un tipo arbitrario que probablemente el verificador de tipos no asignaría. Este tipo de atribuciones no siempre resulta ser una buena idea pues la seguridad del lenguaje se ve comprometida, sin embargo la filosofía a seguir es "confía pero verifica". El verificador de tipos acepta el tipo asignado por la conversión de tipos hacia abajo en tiempo de compilación. Sin embargo inserta una señal que, en tiempo de ejecución, causará que se verifique que el valor tenga realmente el tipo asignado [66].

Una de las ventajas del uso de la unión de tipos graduales, se puede observar en el siguiente programa:

```
1 let  x:Bool (+) Int (+) String = 10      ;;El símbolo (+) denota la suma gradual.
2 ((lambda x:Int (+) Bool.x+1) x)          ;;El símbolo + denota a la suma de valores enteros.
```

Código 2.2: Ejemplo de uso de tipos graduales.

El programa tiene declarada una variable local x la cual puede tener alguno de los tres tipos: booleano, entero o cadena. Al inicializarse ésta con el número 10, el argumento es pasado a la función que espera recibir un entero (en este caso), el cual al sustituirse en el cuerpo de la función se evalúa al entero 11, regresando un valor de tipo entero. Sin embargo, si se recibe como argumento un booleano o una cadena, el programa aunque estaría bien inicializado y regresa un error en tiempo de ejecución antes de ejecutar la suma.

Las uniones graduales de tipos son únicas y permiten ciertos errores en tiempo de ejecución, sin embargo, uno de sus beneficios sigue siendo los patrones de programación flexibles. Los enfoques de tipificado de flujo sensitivo como la ocurrencia de tipos presentan una mayor precisión en la asignación de tipos basándose en los resultados de algún tipo de verificación usando predicados. Este tipo de métodos pueden eliminar ciertas conversiones de tipo innecesarias.

Algunos lenguajes que soportan la unión de tipos (sin etiquetar) frecuentemente también soportan la operación de intersección de tipos, y la relación de distributividad $(T_1 \vee T_2) \to T_3 \equiv (T_1 \to T_3) \wedge (T_2 \to T_3)$. Esta regla permite que una función acepte un valor que puede ser de tipo tanto T_1 como T_2 y éste presente un comportamiento como ambos tipos, ya sea tanto en una función $T_1 \to T_3$ como en otra función de tipo $T_2 \to T_3$. Por otro lado, las uniones graduales comprenden ambos tipos de interpretaciones, sin tener que usar la propiedad de intersección de tipos: $(T_1 \oplus T_2) \to T_3 \equiv (T_1 \to T_3) \oplus (T_2 \to T_3)$. Los autores Castagna y Lanvin han desarrollado otro tipo de enfoque para tipos graduales, conocido como tipos basados en teoría de conjunto (*set-theoretic types*), en el cual se hace uso de las operaciones de unión, intersección y tipos desconocidos. En este sistema de tipos se combinan estas tres operaciones. Por ejemplo $Int \oplus Bool$ es equivalente a $(Int \mid Bool)$ & ? [95].

Para comprender tanto los tipos graduales como la unión de tipos se sugiere utilizar la metodología de la interpretación gradual de unión de tipos. La notación sugerida es $T_1 \oplus T_2$ para definir tipos graduales, el cual abstrae tanto T_1 como T_2. Toro y Tanter exponen el enfoque de asociación de los tipos etiquetados y la unión de tipos sin etiquetar. Para obtener la semántica estática de un lenguaje gradual, el TGA utiliza las conexiones de Galois (entre tipos graduales y los conjuntos de tipos estáticos) para guiar el levantamiento de funciones y predicados de los tipos estáticos a sus tipos graduales.

Resumen

En este capítulo se presenta una introducción a los lenguajes de programación, describiéndolos y exponiendo una visión general de éstos, así como algunas categorizaciones de lenguajes de programación dependiendo diversos enfoques, en particular uno de estos enfoques provee los cimientos del área de investigación relacionada con el tipificado.

Por un lado, se expone un preámbulo a los lenguajes con tipificación dinámica y estática, mostrando las características de cada uno, así como las ventajas y desventajas que presentan éstos. Por otro lado, se introduce el concepto, las características y terminología utilizada en la teoría de tipos o sistema de tipos, así como la definición de algunas de las propiedades de los lenguajes como lo son la seguridad y la consistencia.

Por último se presenta una introducción a los lenguajes con tipificación gradual, para definirlos y exponer sus características. Los elementos y reglas que se exponen dentro de esta clasificación de lenguajes, son requeridos para así construir las bases con las cuales se trabaja en la presente investigación.

Capítulo 3

Trabajo relacionado

> "Hay sólo dos clases de lenguajes de programación: aquellos de los que la gente está siempre quejándose y aquellos que nadie usa."
>
> *Bjarne Stroustrup* [98].

Los lenguajes de programación se pueden clasificar por su tipificado (1) estático y dinámico; por el paradigma al que pertenecen (2) Orientado a Objetos, Funcional, Lógico, Estructurado o bien que mezclen características de varios paradigmas llamados multiparadigma; por el tipo de problemas que resuelven: de propósito general y de dominio específico, entre otras categorías. Bajo cualquier clasificación, los lenguajes de programación son utilizados por los programadores, y algunas veces el desarrollo de software conlleva el uso de varios lenguajes (independientemente de la clasificación en la que se encuentren), por lo que dar una solución adecuada al problema que se desee resolver depende no solo del programador sino de la herramienta (lenguaje de programación en este particular) que se haya escogido [64].

Clasificación de lenguajes de programación por su grado de tipificado

En esta sección se propone una taxonomía de los diferentes enfoques en los que un lenguaje de programación puede representar su grado de tipificado. *Groso modo*, y como ya se había mencionado en el capítulo anterior, los lenguajes pueden ser clasificados por su tipificación, ya sea estática o dinámica; sin embargo, los lenguajes pueden tener diferencias sutiles en el modo en el que se realiza la verificación de sus tipos (usando sus respectivos sistemas de verificación de tipos).

La Figura 3.1 muestra una propuesta de clasificación de lenguajes de programación dependiendo del grado de tipificado que éstos presenten. Dicho de otro modo, se propone aquí una posible taxonomía de los diferentes enfoques en los que un lenguaje de programación puede representar sus grado de tipificado. Los nombres de algunos de los autores principales que han contribuido al estudio de cada una de estas categorizaciones se muestran del lado derecho de la figura.

En la parte superior de la Figura 3.1 se pueden observar las siguientes clasificaciones de lenguajes: lenguajes con tipos dependientes, en los cuales se pueden definir funciones de valores a tipos. En otras palabras, parametrizando una definición de tipo sobre un valor. Más adelante se explicará este concepto. Estos lenguajes presentan dos características importantes: (a) un tipo puede depender de un valor, lo cual (b) permite eliminar un conjunto más amplio de errores en tiempo de ejecución [100]. A su vez algunos de estos lenguajes permiten hacer uso de reglas formales para determinar la dependencia de tipos usando conectivos, tales como: $\wedge, \vee, \leq, \geq, \cap, \cup$, así como los cuantificadores lógicos $\forall$ y $\exists$ [26, 28]. Lenguajes con refinamientos de tipos, permiten enriquecer los lenguajes que los usan. Esto se debe a que el lenguaje permite usar predicados que describen con precisión los conjuntos

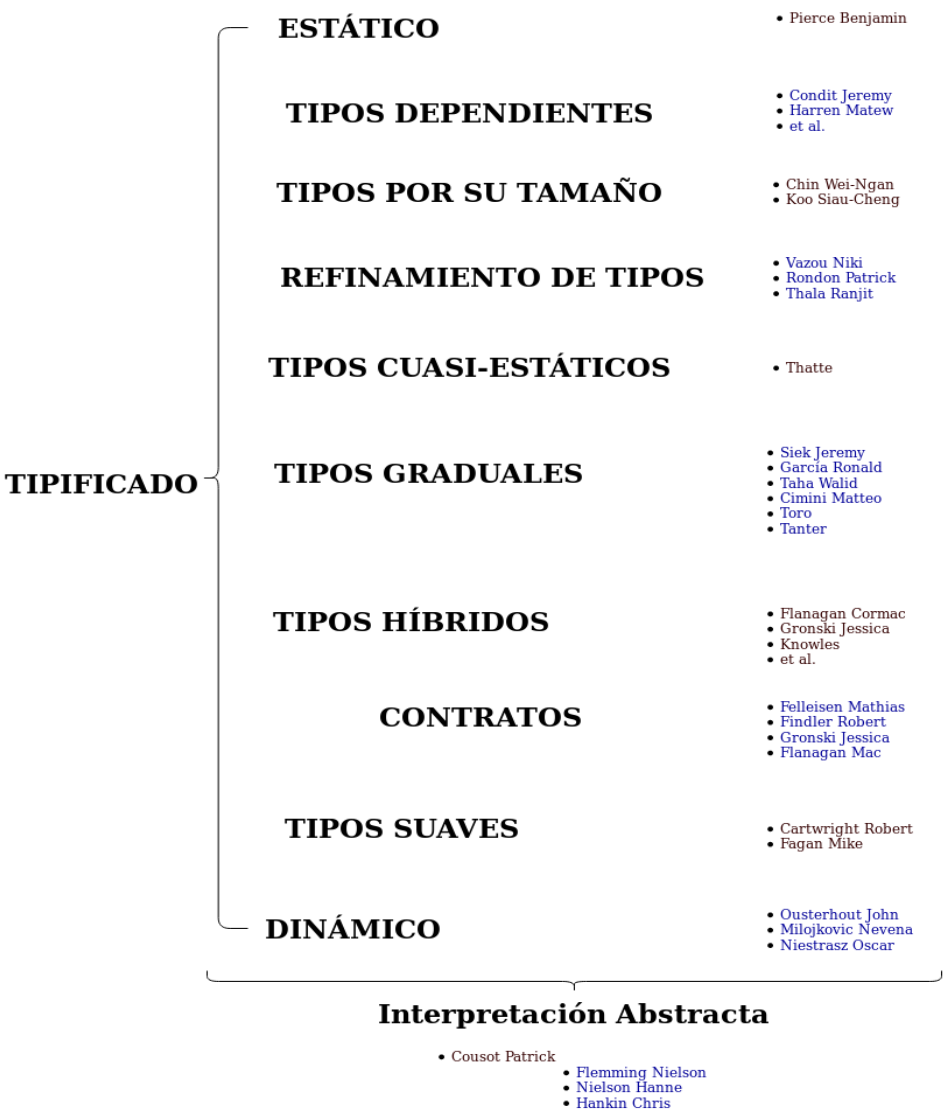

Figura 3.1: Clasificación de lenguajes de programación dependiendo de su grado de tipificado.

válidos de entrada y salida de funciones, valores que se encuentren en contenedores, entre otros. Al combinar tipos con predicados, los programadores pueden especificar contratos que describen entradas y salidas válidas de funciones. Los sistemas de refinamiento de tipos garantizan que en tiempo de compilación las funciones se adhieran a sus contratos [53]. Algunos de estos lenguajes hacen uso de cuantificadores denotados respectivamente por los símbolos lógicos $\forall$ y $\exists$, para así realizar el refinamiento en diversos tipos de datos y funciones presentes en un programa [102]. Los tipos *cuasi-estáticos* requieren declaraciones de tipos explícitas en los argumentos de funciones, no soportan polimorfismo, tipos recursivos y solo se presenta un nivel de subtipificado [90]. Éstos últimos son unos de los primeros esfuerzos por presentar tipos explícitos e implícitos en un mismo lenguaje, aunque su alcance no es tan amplio. Los lenguajes con tipificado gradual [78] permiten al programador añadir etiquetado de tipos cuando éste así lo decida y permiten la omisión de notaciones de tipo también cuando el programador así lo desee. Los lenguajes híbridos, por otra parte, realizan análisis estático cuando es posible y dinámico cuando es necesario. Dicho de otro modo preservan la propiedad de corrección y detectan errores en tiempo de compilación (cuando sea posible) y en tiempo de ejecución (sólo cuando sea necesario). En este tipo de

lenguajes el verificador de tipos añade automáticamente la conversión de tipos para así soportar especificaciones de tipos más precisas [37, 55]. En este particular se han hecho varios esfuerzos para agregar consistencia y seguridad a este tipo de lenguajes, haciendo uso de *contratos* [36], y usando la unificación de tipos híbridos con contratos [45]. Otro ejemplo de ello es el lenguaje *SAGE* el cual utiliza refinamiento de tipos de primera clase y tipos híbridos, junto con el tipo *Dynamic* [46]. En este mismo rubro se ha aportado una metodología para derivar sistemas de tipos graduales, conocido como *The Gradualizer* [25]. La metodología propuesta ayuda a generar semántica estática para lenguajes con tipificado gradual. El término *soft typing* o tipificado suave en español, es usado para programas que no utilizan tipos explícitos en su sintaxis. Una característica de éstos es que el verificador de tipos estático puede dejar pasar programas con errores; sin embargo, el verificador de tipos inserta verificaciones explícitas de tipo en tiempo de ejecución sobre operaciones primitivas, convirtiendo programas con tipificado dinámico en una forma estática [18]. Por último y en la parte inferior de la Figura 3.1 se observa a la *Interpretación Abstracta*, la cual fue propuesta por Cousot [30] que es una metodología para analizar semánticamente un programa independientemente del estilo de la especificación que éste tenga [30]. Esta última parte no pertenece a la clasificación de lenguajes de programación, sin embargo se menciona para hacer explícita una de las metodologías más utilizadas en el análisis de éstos de manera formal. A partir de ésta última algunos autores la han extendido para manejar tipos graduales, conocida como *Tipificación Gradual Abstracta* para determinar la semántica de tipos estáticos pre-existentes en lenguajes con tipificado gradual [42].

A continuación se expondrán algunos de los trabajos más relevantes relacionados con el tipificado gradual en lenguajes de programación.

Tipos dependientes

Un tipo dependiente es un tipo que depende de un valor. Los lenguajes con tipos dependientes permiten parametrizar una definición de tipo sobre un valor. El siguiente ejemplo muestra un pequeño código donde la variable n no solo es dependiente del tipo de la i sino que su valor debe ser menor o igual que ésta.

```
1 type BoundedInt(n)={i:Int | i <= n}
```

Código 3.1: Ejemplo de lenguaje con tipos dependientes [100].

Para verificar el tipo de un programa que utiliza tipos dependientes, el sistema de tipos y el compilador necesitan conocer la semántica de estos valores y sus operaciones. En la Teoría de Tipos intuicionista, los tipos dependientes se usan para programar con cuantificadores lógicos como $\forall$ y $\exists$. Algunos de los lenguajes que utilizan tipos dependientes son Agda, Coq, F*, Epigram e Idris. Los tipos dependientes ayudan a reducir errores permitiendo que el programador asigne tipos que restrinjan aún más el conjunto de posibles implementación. Los tipos dependientes, por un lado, agregan complejidad a un sistema de tipos, y por otro lado permiten mejorar la expresividad de un programa, ya que el compilador puede detectar y "cachar" un mayor número de errores en tiempo de compilación. Esto permitirá que el programador pueda corregir estos errores y mejorar así las posibilidades de que el programa se comporte como se espera en tiempo de ejecución. Un ejemplo de uso de éstos es cuando se tienen contratos.

Los lenguajes con tipificado dependiente ayudan a evitar errores que dependen de algún valor. Por ejemplo, cuando una función recibe cierto valor que podrá ser conocido hasta tiempo de ejecución o bien cuando se usan índices no permitidos de acceder en ciertas estructuras de datos (como arreglos o matrices), es decir, al querer acceder a algún índice inexistente de un arreglo o matriz, las conocidas excepciones de apuntadores nulos (del inglés *null pointer exceptions*), ciclos infinitos, entre otros. Con este tipo de lenguajes podemos expresar casi cualquier cosa. Por ejemplo, la función factorial que solo acepta números naturales, o la función de inicio de sesión que no acepta cadenas vacías, o una función `removeLast` que solo acepta matrices no vacías. Y todo esto se comprueba antes de ejecutar el programa. Ya que el problema con las comprobaciones en tiempo de ejecución es que fallan cuando el programa ya se está ejecutando. Los tipos dependientes permiten realizar estas comprobaciones en el propio sistema de tipos, lo que hace que sea imposible fallar mientras se ejecuta el programa.

Cabe mencionar que los tipos ayudan a especificar invariantes en un programa. Los tipos simples son extendidos por los tipos dependientes para expresar y preservar invariantes en elementos de estado múltiple en

un programa. Por ejemplo, en el manejo de arreglos (1) se debe de preservar el tamaño de éste, (2) se debe de contar con un tipo apuntador del arreglo que señale su inicio y otro apuntador para identificar su límite final, (3) así como algún tipo de identificador que especifique si se encuentra activo o no. El sistema de tipos propuesto por Condit et. al. en [27] es un sistema de tipos dependientes, el cual preserva la propiedad de corrección de un lenguaje. La relación que mantienen los tipos dependientes se establece mediante relaciones basadas en operaciones lógicas, utilizando operadores como $\wedge$, $\vee$, $\leq$ y $\geq$, así como las operaciones entre conjuntos $\bigcup$ y $\bigcap$. Con este tipo de notación en un programa, se pueden establecer ciertas relaciones y restricciones de uso de las funciones, para así garantizar ciertas caraterísticas asociadas a los tipos de éstas usadas en un programa. Este tipo de relaciones, las puede describir un programador al momento de desarrollar un programa y con ello garantizar que tanto en tiempo de compilación como de ejecución los programas se comportan de la manera esperada.

Tipos con dimensiones

Los lenguajes de programación realizan automatizaciones y análisis relacionados con sus tipos. Por ejemplo el tamaño de los parámetros que recibe y regresa una función. Sin embargo, la información sobre el tamaño puede ser difícil de calcular. Un acercamiento a este tema, es realizando una verificación sobre el tamaño de los datos que se manejan y no haciendo inferencia sobre las operaciones primitivas de un lenguaje. La mayoría de las veces, esperamos que los tipos tengan un tamaño positivo y conocido estáticamente. Este no siempre es el caso de algunas estructuras o tipos dinámicos. Un lenguaje con este tipo de tipificado es Rust. Éste admite tipos de tamaño dinámico (del inglés *Dinamically Sized Types, DST*), es decir,tipos sin un tamaño conocido de manera estática [14].

Por otro lado, Chin y Khoo en [23] proponen calcular la información relacionada con los tipos en un programa revisando el tamaño de los datos utilizados en tal programa. En caso de que no se tenga preasignado el tamaño de las variables, funciones, y datos en general resulta difícil realizar algún tipo de optimización y análisis acerca de los tipos en un programa. Exponen un marco de trabajo en el cual se puedan expresar los diferentes variantes de tipos haciendo uso de la inferencia (dado su tamaño) en vez de la verificación de tipos[1].

Chin y Khoo especifican el tamaño de un dato como:

1. El tamaño de un dato recursivo es su máxima profundidad, es decir, el número de veces que se llame (anidaciones). Por ejemplo, el tamaño de una estructura de datos como las listas y los arreglos está dado por su longitud, el tamaño de un árbol está dado por su profundidad, y el tamaño de un entero es el entero *per se*.

2. Para cualquier otro tipo de datos numerado no recursivos, se utiliza la propiedad de tamaño para distinguir un dato de otro. Por ejemplo, los valores booleanos, regularmente se les asigna cero (0) para falso y el valor de uno (1) se escoge para verdadero.

3. El tamaño de una función está determinado por la relación tanto del número y tamaño de los argumentos que recibe, así como del valor de regreso.

Los autores Chin y Khoo presentan las reglas formales y las técnicas que automatizan la inferencia de los tipos de datos utilizados en un programa. Proveen un diseño mejorado del tipificado dado su tamaño (*sized typing*) para realizar optimizaciones y análisis interprocedural, usando restricciones por tamaño (*sized constraints*) e invariantes en funciones recursivas.

Este tipo de tipificación es usada en lenguajes donde es importante mantener un manejo de espacio eficiente, para así poder realizar optimizaciones con los tipos que se declaren y usen en cada una de las funciones y variables de un programa. Lo anterior se puede expresar haciendo uso de la notación de tipificado dependiente, adoptando a su vez las garantías usadas por medio de las reglas formales que permita el lenguaje.

[1] La información del tamaño está expresada en términos de la fórmula de Presburger y el Calculador Omega para calcular lo mejor posible la información relacionada con el tamaño de los datos [23].

Tipos Cuasi Estáticos

Uno de los primeros acercamientos al tipificado gradual, se introduce por Thatte en 1989 con el término **Tipificado Cuasi-Estático**, del inglés *Quasi-Static Typing*. En este enfoque se mezclan en un sistema verificador de tipos tanto el tipificado estático como el dinámico. Este sistema de tipificado conlleva dos fases: la primera que infiere los tipos y realiza el tipificado usando juicios de tipo a través de coerciones, y la segunda fase que identifica si un programa está mal tipificado (*ill-typed*).

El tipificado Cuasi-Estático categoriza a los programas en tres tipos [90]:

Bien tipificados (*Well-typed*): donde el sistema de tipos garantiza que las expresiones definidas en un programa no contienen errores asociados con sus tipos en tiempo de ejecución.

Mal tipificados (*Ill-typed*): en este tipo de programas el sistema de tipos identifica qué programas se encuentran mal tipificados.

Programas ambivalentes: cuando el sistema de tipos en tiempo de ejecución no puede asegurar si un programa siempre se ejecuta con los tipos correctos.

Una manera para tratar de equilibrar ambos enfoques (tipificado estático y dinámico) surge al proponer sistemas de tipos más suaves (*softer type systems*). Esto es, el tener tipificado estático siempre que sea posible y dinámico cuando sea necesario [60]. Desde esta filosofía se construyeron los lenguajes con *tipificado híbrido*.

Tipificado híbrido

Flanagan Cormac en [37] provee un sistema de verificación de tipos híbrida que conjunta lo mejor de los dos mundos (verificación de tipos estática y dinámica), por medio de la verificación de propiedades de consistencia, detectando errores en tiempo de compilación (cuando sea posible) y en tiempo de ejecución (sólo cuando sea necesario). Flanagan profundiza al dar una prueba formal de la relación entre la verificación de tipos híbrida y los enfoques de verificación de tipos, en términos de su expresividad, cubriendo la mayoría de los errores de tipo. De esta forma una verificación de tipos completa presenta ciertas limitaciones en su expresividad (para los programadores). El trabajo de Flanagan muestra un verificador de tipos híbrido que automáticamente añade conversión de tipos para así soportar especificaciones de tipos más precisas.

Cuando la verificación de las especificaciones de un programa[2] usando interfaces se realiza en los lenguajes con tipificado estático, éstas son más efectivas. Sin embargo, cuando se usa verificación dinámica por medio de contratos, estas verificaciones se realizan en tiempo de ejecución dando como resultado una detección de errores incompleta. La verificación de tipificado híbrido es una síntesis de ambos enfoques (estático y dinámico). Por un lado, el enfoque de verificación de tipos estático es usado para poder obtener una verificación completa de un programa con especificaciones restringidas, y por otro lado la verificación de tipos dinámica se centra en una verificación incompleta, promoviendo la expresividad por parte del programador. Ambos enfoques por separado no permiten establecer una solución propicia. Es por lo anterior, que el enfoque de tipificado híbrido contribuye a unir ambos enfoques y así corregir sus limitantes por separado [55].

La verificación dinámica de tipos presenta dos limitaciones:

- Consume ciclos de reloj, es decir, se consumen recursos que pueden usarse en otros cálculos.
- Puede dar resultados incompletos asociados con la detección de errores.

[2]En algunos casos el desarrollo de aplicaciones conlleva el uso de varios lenguajes y junto con éstos el uso de diversas especificaciones de interfaces (también conocido como API). Para verificar que un programa final mantenga las reglas de especificación usadas en cada una de sus partes, se utilizan diferentes técnicas para que el sistema verificador de tipos garantice de manera precisa tales especificaciones o contratos, tanto de manera formal como en la práctica [55].

La verificación de tipos híbrida debe de ser compatible con las reglas de subtipificado (en caso de que se esté utilizando éste). De esta forma, si el verificador de tipos provee y mantiene la relación de subtipificado, entonces se puede decir que el programa está bien tipificado (*well-typed*). Por el contrario, si el verificador de tipos no puede establecer y preservar la relación de subtipificado entonces el programa es rechazado y considerado como mal tipificado [55].

En resumen la verificación de tipos híbrida presenta las siguientes características:

1. Soporta especificaciones de interfaces precisas, promoviendo el desarrollo de software confiable.
2. Detecta tantos errores como le es posible en tiempo de compilación.
3. Acepta todos los programas bien tipificados.
4. Acepta programas que pueden estar mal tipificados.
5. La salida del sistema verificador de tipos híbrido siempre es un programa bien tipificado (en el caso en que se usen las reglas de subtipificado).
6. La conversión de tipos en un programa bien tipificado se mantiene tanto en el código fuente como en su comportamiento.

El tipificado híbrido permite detectar ciertos errores en tiempo de compilación, extendiendo a su vez la verificación dinámica de tipos para soportar especificaciones más precisas. El verificador de tipos decide usando las reglas de subtipificación si un programa se encuentra bien tipificado o no. En particular el tipificado híbrido en un programa falla si al insertar una operación de conversión de tipos (*cast*) $T \triangleleft S$ de un valor v es indecidible si S es subtipo de T [55].

Knowles y Flanagan en [55] mencionan que su sistema de verificación híbrida es parecido a la verificación de tipos gradual; sin embargo, éste último no maneja refinamiento de predicados (*refinement predicates*) y su propuesta sí. Así que el contar con un análisis híbrido con restricciones de refinamiento de predicados ayude a garantizar que ciertos fragmentos de código en un programa pueden ejecutarse sin presentar cierto tipo de errores. Por otro lado, el desarrollo de software es una tarea difícil, sobre todo cuando se habla de sistemas grandes, los cuales requieren que el desarrollo sea modular. Una posible solución es desarrollar especificaciones precisas e interfaces confiables. Sin embargo, en la práctica los programadores desarrollan soluciones basadas en un conjunto de API cuyo comportamiento no necesariamente está especificado de manera precisa formalmente. Para desarrollar software modular es necesario contar con especificaciones precisas. Por lo anterior, la verificación de tipos híbrida es un enfoque que permite desarrollar software de despliegue, en particular cuando se habla de métodos donde ocurre algún problema en la conversión de tipos post-despliegue (*cast failures postdeployment*), mecanismos de *roll-back* transaccionales, entre otros.

Los sistemas de tipos estáticos han sido de gran ayuda cuando se trata de proveer especificaciones formales que garanticen que tanto en tiempo de compilación como de ejecución los sistemas desarrollados tengan un comportamiento predecible. Para facilitar el desarrollo de software donde no se limite a los programadores a ciertas especificaciones establecidas en los sistemas estáticos de tipos, conservando la propiedad de seguridad del tipificado estático, se realiza la verificación dinámica de contratos en las siguientes funciones presentes ya como parte del núcleo de algunos lenguajes [37]:

- Tipos de subrangos, por ejemplo la función *printDigit* que imprime un número entero entre 0 y 9, deberá verificar que éste sea un número entero en un rango del 0 al 9.
- Restricciones de error (*aliasing restrictions*), por ejemplo si se usa la función *swap*, la cual intercambia dos elementros, requiere que sus argumentos se encuentren en distintas celdas de memoria.
- Restricciones de orden, por ejemplo en la función *binarySearch* la cual requiere que los argumentos se encuentren en un arreglo ordenado, para así realizar la búsqueda binaria de algún elemento.
- Especificaciones de tamaño, por ejemplo en la función *serializeMatrix* la cual requiere una matriz de tamaño n por m, y así realizar la serialización de una matriz dada.

- Predicados, se requiere que el código esté bien tipificado, es decir, que satisfaga el predicado *wellTyped:* $Expr \implies Bool$.

Al unir ambos planteamientos, es decir, tanto la verificación de tipos completa y un nivel alto de expresividad en las especificaciones, se da origen a la verificación de tipos híbrida. Con el fin de coadyuvar al desarrollo de sistemas de tipos expresivos, el sistema de tipos difícilmente puede clasificar estáticamente a los programas como bien tipificados y mal tipificados, se dice que los programas son [37]:

- Rechazados estáticamente: programas rechazados por el compilador, aunque algunos de éstos pudieran ser categorizados como bien tipificados usando un análisis más detallado.
- Aceptados estáticamente: programas que pueden violar alguna especificación en tiempo de ejecución.

La verificación de tipos de los lenguajes con tipificado dinámico se puede hacer usando (1) inferencia de tipos, (2) contratos, (3) subtifipado, (4) tipos genéricos, (5) covarianza [3], (6) relaciones *ad-hoc* y herencia [60].

Gronski y Flanagan en [45] exponen que los lenguajes con tipificado híbrido presentan la suficiente expresividad para que los programadores puedan desarrollar aplicaciones de manera rápida (prototipos). Sin embargo este tipo de lenguajes presentan a su vez algunas desventajas como el no poder garantizar o refutar algunas propiedades en sus programas, tal y como lo hacen los sistemas verificadores de tipos estáticos. Desarrollar un sistema de verificación de tipos híbrido que permita decidir qué módulo (o función, dependiendo sea el caso) puede ser el responsable de evaluarse con el tipo adecuado tanto en los parámetros que reciba como en su tipo de valor de regreso. Para ello proveen un sistema de tipos que combina el tipificado híbrido y el uso de contratos en un lenguaje. Gronski y Cormac proponen un sistema de tipos híbrido λ^C junto con un sistema de contratos λ^H, dando un enfoque formal que permite ver la relación entre ambos enfoques. Por otro lado Gronski et al. exponen que los verificadores de tipos de programas escritos en lenguajes sin tipificado explícito pueden aceptar programas que en tiempo de ejecución violen algunas reglas de tipificado de manera estática. Su propuesta es desarrollar un lenguaje, SAGE [4], el cual utiliza tipificado híbrido combinando el uso de refinamiento de tipos junto con tipos de primera clase y el tipo *Dynamic*.

La verificación de tipos que utiliza SAGE muestra un algortimo para verificación de tipificación híbrida, que cubre las limitaciones de decibilidad, si un lenguaje preserva en tiempo de ejecución sus tipos de manera correcta (bien tipificados), si es rechazado (mal tipificado) o si se desconoce si éste se encuentra bien o mal tipificado. Este algoritmo hace uso de una base de datos en la cual se van almacenando tanto las expresiones definidas en un programa como los tipos asociadas a cada una de éstas [5]. Cuando el algoritmo no pueda garantizar que en una expresión el tipo de una expresión sea subtipo de otro entonces SAGE almacena en su base de datos de contraejemplos aquellas expresiones que no cumplan con que S sea subtipo de T. De tal suerte, que el algortimo antes de rechazar o aceptar una expresión, consulta en la base de datos (en tiempo de compilación) si una expresión en el programa que está analizando corresponde con alguna expresión antes almacenada donde se verifique que S sea subtipo de T.

Tipificado híbrido dentro del Paradigma de la Orientación a Objetos

Las características más importantes de este tipo de lenguajes son la metaprogramación, la reflexión, la movilidad y la reconfiguración dinámica. Ejemplos de este tipo de lenguajes son (1) Ruby, con el desarrollo de aplicaciones Web haciendo uso del marco de trabajo Ruby On Rails; (2) Java Script, también usado en desarrollo de aplicaciones Web mediante AJAX (*Asynchronous JavaScript And XML*); (3) PHP (*Hypertext Preprocessor*) el cual es uno de los lenguajes más populares para desarrollar Web con contenido dinámico; (4) Python, en particular con el marco de trabajo Web Django para desarrollo de aplicaciones, así como el servidor de aplicaciones Zope y (5) C# el cual aún siendo un lenguaje con comprobación estática de tipos ha agregado la comprobación dinámica usando el tipo dynamic como parte del núcleo de su lenguaje; entre algunos otros lenguajes. La principal ventaja de la comprobación dinámica de tipos brinda una posible solución al almacenamiento de datos de manera persistente, la comunicación entre procesos, y la creación dinámica de programas, así como la manipulación del

[3] La covarianza combina el subtipificado paramétrico y el polimorfismo de subtipos [60].
[4] Este término es usado para plantas del género Salvia y es el nombre de este lenguaje de programación [101].
[5] La base de datos se va actualizando con cada verificación de algún programa.

comportamiento de éstos. Por otro lado, la comprobación estática posibilita la detección temprana de errores de tipo, teniendo una mejor documentación del código, y proporcionando optimizaciones que realiza el compilador, mejorando así el rendimiento de los programas [39].

García y Ortin en [39] proponen el diseño de un lenguaje híbrido, StaDyn dentro del paradigma de la Orientación a Objetos, el cual es una extensión de C# 3.0[6]. El lenguaje StaDyn extiende el comportamiento del tipificado implícito de variables locales de C# 3.0, por lo que el tipo de referencias puede ser usado de manera tanto explícita como implícita usando en este último caso la palabra reservada `var`, haciendo uso del algoritmo de Hindley-Milnder en la inferencia de tipos en este tipo de variables. En los lenguajes dinámicos se define la propiedad **duck typing**[7] como [39]:

> "La característica de que un objeto sea intercambiable por otro objeto que implemente la misma interfaz dinámica, independientemente de si tienen una relación de herencia o no".

StaDyn a su vez presenta la propiedad de *duck typing* con comprobación estática de tipos. Este lenguaje posibilita el uso de referencias tanto estáticas como dinámicas de `var`. Si la comprobación e inferencia de tipo es pesimista entonces la comprobación es estática, y es optimista si se hace de manera dinámica, sin modificar la semántica dinámica del lenguaje de programación. Los enfoques de tipos cuasi-estáticos, Tipificado Híbrido (TH) y Tipificado Gradual (TG) realizan una conversión implícita de tipos entre código dinámico y estático, haciendo uso (en el caso de las dos primeras) de las relaciones de subtipificado algorítmico y de la propiedad de consistencia (corrección) en el caso del tipificado gradual. Con el objetivo de añadir comprobación estática y dinámica tanto el TG así como el diseño del lenguaje StaDyn plantean la resolución de restricciones haciendo uso de la unificación [39].

Tipificado suave

El uso de lenguajes de *scripting* se ha incrementando a lo largo de los años. La mayoría de éstos presentan un sistema de verificación de tipos dinámica, por lo que estudiarlos resulta una tarea interesante y actual. Así, los lenguajes con tipificado estático y dinámico presentan ventajas y desventajas tanto para el programador como para el compilador o intérprete del lenguaje. Los programadores deben de usar aquellos lenguajes para los cuales resulte más adecuado resolver un tipo de problemas en particular, tomando en cuenta el tipificado que presenten [64].

Cartwright y Fagan en [18] proponen asignar de manera dinámica los tipos de los valores en un programa en un esquema de tipificado dinámico, preservando algunas propiedades de sus operaciones tanto de manera estática como dinámica. Proveen un sistema de tipos en los lenguajes que no presentan tipos explícitos, para así garantizar las propiedades de consistencia y seguridad tanto en tiempo de ejecución como de compilación.

Se han realizado esfuerzos relevantes para mostrar que el diseño de este tipo de lenguajes es satisfactorio, en el sentido de que los tipos especificados en la parte formal se preserven en cada una de las expresiones definidas tanto por el programador como por el código generado en tiempo de ejecución [60].

El tema de la inferencia de tipos se relaciona con el problema de asignar tipos a cada una de las expresiones en un programa en lenguajes no tipificados explícitamente. La inferencia de tipos pretende restaurar declaraciones de tipo omitidas tanto para las variables como para las funciones. En el contexto de las matemáticas, el problema reside en dar una solución a un sistema de ecuaciones sobre el álgebra de tipos no interpretado, cuyas variables no presenten declaraciones de tipo [18, 93].

La inferencia de tipos puede presentar alguno de los siguientes problemas [93]:

- Fragilidad sintáctica: el hacer una pequeña modificación al código, la cual conserva la semántica de alguna expresión, puede convertir una especificación en una descripción originalmente de dos líneas en una de diez líneas.

[6] Este lenguaje presenta tanto comprobación estática como dinámica. En este lenguaje tanto las características como la consistencia y eficiencia de un lenguaje con comprobación estática de tipos se preserva, así como la flexibilidad y adaptabilidad de los lenguajes con comprobación dinámica [39].

[7] En este trabajo se usa el nombre de la propiedad en inglés *duck typing* y no su traducción en español *tipificado de pato* debido a su poca adaptabilidad al realizar la tradución en este texto.

- Errores no ejecutados: esto se refiere a que no en toda ejecución un programa ejecutará necesariamente todas las líneas de código provistas en un programa, ya que esto dependerá de las instancias con las cuales se esté ejecutando un programa.
- Pérdida de modularidad: al utilizar el algoritmo de Hindley-Milner solo se presenta el tipo de error en el lugar donde se origina, sin tomar en cuenta el análisis completo del programa.

> "En el contexto de lenguajes con tipificado migratorio, el cual se define como una instancia particular del tipificado opcional para lenguajes con tipificado implícito[8], es fundamental mantener su consistencia o corrección. Dicho de otro modo, se desea que este tipo de lenguajes presenten la consistencia de los lenguajes fuertemente tipificados, y a su vez puedan interactuar en ambientes donde se le permita al programador tener tipos explícitos o implícitos" [93].

De lo anterior se han formalizado dos teoremas acerca de la consistencia de estado (*Soundness Theorem States*). Conocidos por sus siglas MT1 y MT2, donde MT proviene del inglés *Migratory Typing*, el primer teorema (MT1) hace referencia a los tres primeros puntos que a continuación se mencionan, y el segundo (MT2), si el lenguaje cumple el cuarto punto. Este último punto (de MT2) es utilizado para generalizar programas donde se mezclen ambos tipos de tipificado (*mixed-type programs*)[9]:

1. La ejecución de un programa, termina ya sea regresando o imprimiendo un valor del tipo esperado.
2. La ejecución diverge.
3. La ejecución termina en algún estado de un conjunto de excepciones bien definidias.
4. La ejecución termina en un estado excepcional y ésta se encuentra entre un pedazo de código tipificado y no-tipificado.

Actualmente los lenguajes con tipificado migratorio, conocidos también con el nombre de (1) tipificado híbrido y (2) tipificado gradual satisfacen las cuatro características de la propiedad de consistencia citadas en [93].

> "El tipificado híbrido tiene como objetivo aumentar la potencia del verificado estático. Este permite a los programadores agregar predicados arbitrarios a las especificaciones de tipo, que se comprueban estáticamente si es posible, y dinámicamente en otro caso. El tipificado gradual tiene como objetivo poner los tipos estáticos y dinámicos en el mismo nivel de igualdad sin violar la consistencia."

Tipificado nominal y estructural

Kühne unifica dos enfoques de tipificado en lenguajes (1) el nominal y (2) el estructural. Estos enfoques presentan ciertas diferencias en la manera en que se modelan y desarrollan sistemas dentro del marco de la Ingeniería de Software en el paradigma de la Orientación a Objetos. La ocurrencia de tipos en lenguajes de programación conlleva ciertas características, es decir, atributos que representan propiedades, y operaciones que modelan su comportamiento. Dentro del paradigma de la Orientación a Objetos, se define a un objeto como la instancia de un tipo y, un tipo es un subtipo de un supertipo. Un tipo se denota por T. A este concepto se le asocia (1) intensión y (2) extensión. Por ejemplo, la intensión del concepto *pato* puede ser tomada como una lista de requerimientos, y su extensión como un conjunto (en principio de un solo individuo pero puede incremetarse en número). Este concepto es usado para definir el tipo *pato* o en inglés *duck typing*. Este concepto es asociado al principio de que *si camina como pato y se comporta como pato, entonces es un pato*. El *duck typing* representa una forma muy extrema de verificar la compatibilidad de tipos aplazando tal verificación hasta tiempo de ejecución [58].

El tipificado estructural asume de manera estática los tipos conocidos y las relaciones entre las instancias y sus tipos. En este enfoque los nombres de los tipos se utilizan con propósitos de comunicación. Kühne utiliza el

[8] El primer lenguaje de este tipo es *Common LISP* [93].

[9] Esta idea también es expuesta por Krishnamurthi en su libro Programming Languages. Application and Interpretation [56] sin darle como tal dicho nombre al teorema.

término **tipificado descriptivo** en vez de tipificado estructural para evitar la posible confusión asociada a la palabra *estructural*, pues ésta se extiende no solo a la estructura del objeto, sino también a su comportamiento.

El tipificado descriptivo permite que la pertenencia de una instancia dependa sólo de si presenta o no las propiedades definidas. Un ejemplo de ello es la taxonomía clásica de los organismos basada en sus características observables. Formalmente, todos los miembros de un conjunto se define como[10]:

$$T.\varepsilon_d = \{x \mid T.l(x)\}$$

El símbolo para describir la propiedad de extensión está denotado por $T.\varepsilon$ define al conjunto de instancias que son miembros del concepto T. Por otro lado el símbolo para describir la intensión es $T.l$ y ésta hace referencia a una expresión lógica, la cual envuelve predicados que caracterizan cuando una instancia se encuentra dentro del concepto T. La siguiente regla describe la relación que existe entre la intensión y la extensión: $(T_1.l \sim T_2.l) \rightarrow (T_1.\varepsilon_d = T_2.\varepsilon_d)$ es decir el resultado de la intensión y la extensión son idénticas [58].

El tipificado descriptivo presenta los siguientes requerimientos:

R1: los objetos que se ajustan o que cumplen un concepto deben ser clasificados fácilmente.

R2: no se deben capturar instancias que no se ajusten a algún concepto.

R3: el costo asociado con el mantenimiento de la taxonomía que se genere debe ser tan pequeño como sea posible.

El tipificado nominal se refiere a los identificadores utilizados para dar un nombre a algún tipo. El sistema de tipos nominal brinda explícitamente los nombres de los tipos y el subtipificado debe de ser explícito. Los tipos nominales llevan consigo la intensión implícita, es decir, si un objeto se define como de algún tipo, éste automáticamente se compila a este tipo, cumpliendo la siguiente propiedad $\forall x : (x.type = T) \rightarrow T.l(x)$.

El tipificado nominal presenta los siguientes requerimientos:

R4: los tipos deben de poder distinguirse incluso en la ausencia de las diferencias fenomenológicas entre instancias.

R5: la sustitución de tipos debe ser otorgada por los modeladores.

R6: los tipos deben de ser definidos tanto explícita como sosteniblemente en términos de su creación y costo de mantenimiento.

Al unificar tanto el enfoque nominal como el descriptivo en un marco de trabajo se preservan las caracterísiticas de ambos. Por un lado el enfoque nominal asegura que todos los tipos nominales son distinguibles unos de otros, y por otro lado, los tipos descriptivos son equivalentes unos con otros, a menos que los criterios que los diferencían estén explícitamente establecidos.

Sin embargo, los objetos no ligados son objetos regulares desde un punto de vista descriptivo. Estos objetos siempre existen o son descubiertos en cierto punto, pues las reglas de tipificado descriptivo se les aplican en algún momento. Desde el punto de vista nominal, estos objetos que no están ligados son conocidos como *huérfanos*. Un modelador constructivo, trata de categorizarlos de acuerdo a sus características, mientras permanezcan en este estado de no ligados, se les conoce como objetos en *cuarentena* [58].

Dado lo descrito anteriormente, el marco de trabajo que desarrolla Kühne es:

1. Todos los tipos pueden ser vistos desde dos enfoques: (1) enfoque descriptivo (a través de su intensión) y por su caracterización (para mantener su extensión).

2. Los objetos descubiertos pueden ser categorizados como objetos adoptados (adopción de objetos) asignándoles ciertos tipos de caracterizaciones (conocidas también como tipos de encarnaciones o por el tipo de su descendencia).

[10]Kühne usa la notación $T.\varepsilon$ en vez de $\varepsilon(T)$ y el subíndice d es para denotar el tipo descriptivo. La intensión de $T.l$ es libre para adquirir cualquier propiedad, incluyendo comportamiento [58].

3. La clasificación consolidada sólo considera los tipos caracterizados o bien los tipos descriptivos dependen del estatus de algún objeto en cuestión, es decir, si se encuentra ligado a otro tipo descriptivo.

4. El consolidar el subtipificado solo distingue entre objetos ligados y no ligados. En el enfoque nominal, estas dos categorías de objetos representan a los *ciudadanos certificados* y los segundos a los *huérfanos*.

La idea del tipificado gradual se encuentra relacionada con el *duck typing*. Ambas ideas están de algún modo relacionados con el tipificado estructural. Se consideran las propiedades de cada instancia en vez de verificar su procedencia. Tanto el tipificado nominal como estuctural tiene un enfoque estático, es decir, que la verificación de tipos se realice en tiempo de compilación. Es posible integrar los conceptos de tipificado gradual dinámico en un enfoque unificado. Así el tipificado estructural debe ser posicionado entre los extremos del tipificado dinámico y nominal.

Tipificado gradual: características

Los lenguajes que combinan la verificación de tipos estática y dinámica son conocidos como lenguajes con tipificado gradual. Bajo este estilo de tipificado, los tipos que son verificados de manera estática ayudan a prevenir ciertos errores asociados al comportamiento (en tiempo de ejecución) de un programa. Por un lado, garantizan que el programa se comporta de acuerdo a sus tipos explícitamente definidos, lo cual conlleva el detectar cierto tipo de errores de manera temprana, generando así código de máquina más eficiente. Por otro lado, el tipificado dinámico ayuda a generar desarrollos rápidos de prototipos de software, donde a su vez la programación sea más flexible para los programadores; sin embargo, el tipificado dinámico suele consumir más recursos computacionales y la detección de errores no resulta ser una tarea fácil de realizar. En resumen, se puede decir que ambos enfoques se complementan [48, 51].

Siek y Taha en 2006 definen la propiedad de consistencia de tipos en las expresiones de un lenguaje con tipificado gradual tanto de manera estática como dinámica, la cual se define a partir de funciones parciales. Siek y Taha son los primeros en introducir el concepto de **Tipificación Gradual, (TG)**. En su propuesta se combina la verificación de las propiedades de tipos estáticos y dinámicos.

El tipificado gradual cubre tanto los beneficios de la verificación estática como de la verificación dinámica en un lenguaje, proporcionando al programador el poder de decisión acerca de qué partes de un programa pueden ser verificadas estáticamente y cuáles de manera dinámica, es decir, a qué variables del programa le pueden añadir tipos y a cuáles no [79].

Ventajas que presenta el Tipificado Gradual [42, 79]:

1. Se tiene presente tanto el tipificado estático como el dinámico en un programa, obteniendo entonces los beneficios de ambos enfoques.

2. El programador tiene un control más fino acerca de los tipos de cada expresión en el programa, flexibilizando la programación, es decir, el programador puede o no escribir los tipos de manera explícita en cualquier expresión de un programa.

3. Los programas que se rechazan en tiempo de ejecución mantienen las mismas propiedades de seguridad de manera estática.

4. Los programadores pueden omitir ciertas anotaciones de tipo y el verificador de tipos debe garantizar su seguridad.

5. Los programadores pueden añadir anotaciones de tipo para así incrementar la verificación estática y con ello todos los errores de tipo se recaban en tiempo de compilación.

6. El sistema de tipos y la semántica deben minimizar la carga de implementación en los programadores del lenguaje.

El fundamento teórico del Tipificado Gradual se cimenta bajo algunos de los principios del Tipificado Gradual Abstracto (*Abstracting Gradual Typing, AGT*), el cual propone la semántica de estos lenguajes en términos de tipos estáticos pre-existentes [42].

La diferencia con otros enfoques es que el Tipificado Gradual propone una mezcla entre verificación de tipos estática y dinámica en un programa, con el objetivo de que el programador decida de forma indirecta (al no incluir anotaciones de tipo) qué partes de un programa son verificadas de forma estática y cuáles de forma dinámica. Cabe resaltar que es el compilador quien realizará tales verificaciones. Los lenguajes tipificados gradualmente presentan tres propiedades [81]:

1. Los programas pueden prescindir de ciertas anotaciones de tipos y el programa, aún así se debe ejecutar de tal forma que la verificación de tipos sea segura y consistente en tiempo de ejecución.

2. Los programas pueden incorporar anotaciones de tipos para así extender la verificación de tipos estática. De esta forma, todos los errores asociados con tipos serán atrapados en tiempo de compilación, si todas las variables tienen asociado su tipo.

3. Las partes de un programa que sean tipificadas estáticamente, no deben enviar errores de tipo en tiempo de ejecución.

Una de las características del tipificado gradual es su capacidad de coerción, por medio de la cual en un programa se permite, por ejemplo, que en una expresión que recibe un valor de tipo *unknown* se convierta en un valor de tipo entero estática y dinámicamente. Se dice entonces que un programa donde se utilice una conversión de tipos (*cast*) y ésta sea segura, entonces existe una coerción segura (*safe coercion*). Al presentarse esta condición de coerción, un programador puede migrar un programa con tipificado dinámico a uno con tipificado estático. Si un programador utiliza un lenguaje con Tipificado Gradual, entonces éste puede añadir anotaciones de tipos, de tal manera que el compilador de dicho programa pueda agregar o no las verificaciones asociadas al tipificado, tanto en tiempo de compilación, como en ejecución [81].

La relación de consistencia en el tipificado gradual se denota por el símbolo ~. La idea general de este tipo de relación es verificar si dos tipos son consistentes en cada una de sus partes y está definida para los tipos de las funciones en un lenguaje [11]. Cabe resaltar que esta relación debilita la igualdad de tipos cuando se presenta el tipo *unknown* en alguna de sus partes o lados [42, 81]. La relación se define como sigue:

1. Reflexiva: $\dfrac{}{T \sim T}$
2. Simétrica: $\dfrac{}{T \sim ?} \qquad \dfrac{}{? \sim T}$
3. No transitiva: $\dfrac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}$

En general, el poder combinar ambos enfoques en uno solo ha sido una tarea complicada; sin embargo, actualmente se tienen varias propuestas para poder sacar el mejor resultados conjuntando ambas filosofías.

Los sistemas de tipos graduales abstractos ofrecen una vía que conecta los programas tipificados dinámica y estáticamente, para así obtener códigos donde se combinen los tipos explícitos o implícitos. Herman, Tomb y Flanagan en [48] proponen una semántica para Tipificado Gradual basada en coerciones, en la cual se conjuntan coerciones adyacentes en tiempo de ejecución para limitar el manejo de espacio.

La propuesta de Herman et al. provee la detección de errores en lenguajes con tipificado gradual de manera eficiente utilizando coerciones usando el sistema de tipo propuesto por Siek y Taha del Cálculo Lambda para el tipificado gradual $\lambda^{?}_{\to}$. Este enfoque permite detectar errores de manera temprana en lenguajes con tipificado gradual, lo cual fomenta el tener la máxima flexibilidad en la programación detectando los errores tan pronto como sea posible.

La tabla 3.1 resume algunas de las principales características de los lenguajes con tipificado híbrido y gradual. En la primer columna se señalan los autores que modelan conversiones de funciones de orden superior; la segunda columna se refiere al realizar el seguimiento preciso de la culpa [12]. La siguiente columna señala si en estos

[11] Se usará el tipo T para denotar una metavariable que abarque cualquier tipo arbitrario.

[12] Del inglés *precise blame tracking*.

enfoques se presenta el tipo dinámico *Dyn*; le sigue el uso de de predicados. Por ejemplo, $\{x : Int \mid x >= 0\}$ el cual denota a los números enteros no negativos; la detección de conversión de composición de funciones en tiempo de ejecución, antes de que una función sea aplicada. Y finalmente si se garantiza o no un manejo de espacio eficiente. En la última columna se señala si existe la representación de conversión de funciones, coerciones explícitas, *cast* explícitos o bien se usa *threesomes*. Los sistemas basados en contratos crean envolturas de funciones *proxies*. En estos contratos se verifican y deben de mantenerse tanto los argumentos de las funciones como sus respectivos resultados, de manera perezosa. La principal contribución de este trabajo es el mecanismo de cómo se da seguimiento a las funciones que resulten *culpables* de alguna falla en el programa. Los sistemas de tipos híbridos utilizan las conversiones de orden superior para superar las limitaciones de la decibibilidad y expresividad de los sistemas de tipos tradicionales. Los primeros trabajos que se presentaron en este campo son una extensión del Cálculo Lambda con tipificado simple con predicados de tipos, donde existen tipos base o básicos, como lo es un entero. Utilizando los tipos de funciones dependientes con los predicados de tipos se obtuvo una especificación para manejo de interfaces de funciones, lo cual ayuda a facilitar el desarrollo de software modular. Los trabajos subsecuentes han extendido esta filosofía, incluyendo por ejemplo el tipo dinámico *Dyn*, esclareciendo la metateoría y extendiendo el soporte para poder dar seguimiento al Teorema de la Culpa, así como otras propiedades [48].

El Cálculo de la Culpa soporta la integración de código tipificado estática y dinámicamente. Este cálculo conjunta ambas clasificaciones de tipificado usando la conversión de tipos. Esto se puede realizar compilando al Cálculo de Coerciones y posteriormente se usa el Cálculo *Threesome* [77].

Publicación/Implementación	Conversiones de Orden Superior	Teorema de la Culpa	Tipo Dyn	Predicados de Tipos	Verificación Temprana Errores	Manejo Espacio Eficiente	Representación de CAST
Sistema de Contratos							
Findler y Felleisen	X	X					Función
Sistemas de Tipos Híbridos							
Flanagan	X			X			Función
Knowles y Flanagan	X			X			Función
Gronski et al.	X		X	X			Función
Greenberg et al.	X	X		X			Cast
Sistemas de Tipos Graduales							
Siek y Taha	X		X				Cast
Herman et al.	X		X		X	X	Coerción
Siek et al.	X	X	X		X	X	Coerción
Siek y Wadler	X	X	X		X	X	Threesome

Tabla 3.1: Comparación de sistemas de tipos híbridos y graduales [48].

Por otro lado, Siek y Taha usan envolturas de orden superior permitiendo la interoperabilidad entre códigos con tipos explícitos e implícitos, enfocándose en los lenguajes funcionales y los Orientados Objetos. Herman

et al. en [48] utilizan el concepto de coerción de Henglein para proponer una representación en tiempo de ejecución eficiente, en el manejo de conversiones de orden superior. Por otro lado, Siek y Taha introducen el concepto de *threesome* para reemplazar una secuencia normalizada de coerciones. En particular, un *threesome* $(T_3 \Leftarrow T_2 \Leftarrow T_1)$ es esencialmente una conversión hacia abajo *(down cast)* del tipo T_1 en el tipo T_2 seguido de un *cast* hacia arriba *(up cast)* de T_2 a T_3. Hay que notar que el *threesome* no es equivalente a una conversión de tipos convencional, donde $(T_3 \Leftarrow T_1)$ de T_1 a T_3, ya que la conversión intermedia del *down cast* a T_2 puede fallar en algún valor de sus argumentos. Esta representación de *threesome* corresponde a coerciones normalizadas.

En un lenguaje con tipos gradual se pueden identificar tres partes [79]:

1. El lenguaje de tipificado gradual por sí mismo.
2. Un lenguaje de tipificado estático (semántica estática).
3. Un lenguaje que especifica la semántica dinámica con respecto a las conversiones de tipo.

Cimini y Siek en [25] exponen tanto una metodología, como un algoritmo para generar un sistema de tipos gradual para un sistema de tipos bien formado, manteniendo la concordancia del compilador con el cálculo de conversión de tipos (*Cast Calculus*). La metodología propuesta sirve para generar una semántica estática para lenguajes tipificados gradualmente. Esta metodología es aplicable a la mayoría de los sistemas de tipos propuestos en este trabajo, los cuales son extensiones de *Simply Typed Functional Language, STFL.* el cual, está a su vez presentado y estudiado a fondo por Pierce [66] *Simply Typed Lambda Calculus, STLC*, que está extendido para pares, tuplas, operadores de ligado (`let` y `letrec` de manera recursiva), recursión (`fix`), excepciones, referencias, listas, expresiones `if then else`, entre otras. El algoritmo propuesto es llamado *El Gradualizador*, el cual genera un sistema de tipos gradual y un compilador de un sistema de tipos estático[13].

Garcia y Cimini en [41] proponen el diseño de un sistema de inferencia de tipos gradual que produzca tipos estáticos. El enfoque de estos autores está basado en analizar los programas (con tipificado gradual) verificando que sean consistentes en términos de un sistema de tipos pre-existente. Con base en trabajos anteriores los autores García y Cimini proveen el diseño de un sistema de inferencia de tipos gradual y un algoritmo de inferencia, el cual garantiza la consistencia de los programas con tipificado implícito.

El término *tipificado implícito* en dicho estudio significa que:

> "El lenguaje se encuentra tipificado estáticamente, sin embargo los programadores pueden omitir la anotación de tipos" [41].

La solución que proponen está centrada en la consistencia de los programas más que en el desempeño de éstos. Para ello los autores proveen:

- Una especificación formal de tipificado implícito estático centrándose en las entradas y salidas de un sistema de tipos. Agregando las reglas para utilizar polimorfismo de *let* usando *Polymorphic Blame Calculus.*
- Las soluciones y restricciones de los juicios de tipo respectivos al sistema de tipos diseñado.

El mayor desafío del tipificado gradual es añadir tipos a los programas sin requerir que se hagan cambios significativos en su implementación. Esta postura resulta crítica cuando se trata de añadir Tipificado Gradual a sistemas de tipos preexistentes [89].

Lo anterior se propone con el fin de mostrar que los programadores pueden mezclar de manera segura ambos enfoques sin presentar una sobrecarga en manejo de espacio. Lenguajes como LISP, Scheme, Smalltalk y JavaScript son lenguajes con tipificado dinámico, usados regularmente en el prototipado de soluciones rápidas, cuyo desarrollo de soluciones (sistemas) se hace de manera incremental, usualmente modificando las invariantes estructurales como son los tipos. Los tipos estáticos son cruciales para entender y reforzar las principales invariantes y abstracciones de un programa, *atrapando* tantos errores como sean posibles lo más temprano que se puedan durante el proceso de desarrollo. Debido a las diferencias intrínsecas de cada clasificación de lenguajes (tipificados dinámica o estáticamente), resulta fácil encontrar el siguiente escenario: un programador comienza a desarrollar un prototipo (aplicación) en un lenguaje con tipificado dinámico, donde de manera paralela o por

[13]La propuesta está hecha en Haskell y produce un verificador de tipos y un compilador en Lambda-Prolog [25].

separado, con un equipo o de manera individual. Dicho desarrollo va creciendo, convirtiéndose en un sistema puesto en un ambiente de producción completo y grande. Es posible que con el paso del tiempo este sistema resulte difícil de mantener, pues no presenta una estructura sólida, y no garantiza ciertos resultado asociados a los tipos utilizados. El sistema resultante entonces es pesado y los errores difíciles de resolver, reproduciéndose con rapidez. Dada la situación descrita anteriormente se han establecido mecanismos para mezclar las ventajas que presentan los lenguajes con tipificado dinámico y estático a la vez, por ejemplo en Boo, VisualBasic.NET, Sage, Clean y Racket. Este enfoque de tipificado híbrido ha sido nombrado por Taha y Siek como tipificado gradual [48].

Algunos lenguajes que soportan tipificado gradual son Cecil, Boo, algunas extensiones tanto para Visual Basic.NET como para Java, así como C# y Bigloo [78]. El tipificado gradual introduce el tipo **desconocido** (*unknown type*) usando el caracter ?, así como la relación de *consistencia de tipos* (*type consistency*) en un sistema de tipos estático pre-existente. Los conceptos anteriores ayudan a soportar la verificación tanto estática como dinámica de tipos [42].

Algoritmo de inferencia de tipos para Tipificado Gradual

Como ya se ha expuesto el tipificado gradual ayuda a los programadores a convertir sus códigos no tipificados explícitamente en programas tipificados estáticamente, obteniendo los beneficios de las anotaciones de tipos explícitos en desempeño y en la detección de errores de tipo de manera temprana. Sin embargo, este tipo de lenguajes presentan algunos problemas al permitir omitir ciertas anotaciones de tipos. Lo anterior provoca la introducción de los tipos dinámicos o desconocidos (*dynamic* y *unknown* respectivamente). Rastogi et al. presentan un algoritmo para hacer inferencia de tipos en un lenguaje con tipificado gradual que mejore el desempeño de estos programas, sin introducir nuevos errores de tipos en tiempo de ejecución [14] [69].

Rastogi, Chaudhuri y Hosmer proponen un algoritmo para dar una posible solución al problema interno de retroalimentación de la inferencia de tipos en un lenguaje con tipificado gradual, dicho problema se refiere al hecho de que los programadores pueden o no tipificar algunas partes de código, incluso pueden declarar tipos (por ejemplo en los tipos de los argumentos de funciones) que no satisfagan las restricciones del sistema de tipos estático en el cuerpo de dichas funciones. El algoritmo que proponen infiere los tipos estáticos donde sea posible para garantizar la propiedad de seguridad y tener la oportunidad de hacer algunas optimizaciones sobre éstos. Este algoritmo permite reducir las imprecisiones asociadas a los tipos dinámicos. La inferencia realizada sobre cada variable es la abstracción de un tipo a partir de un conjunto de valores que fluyen hasta éste. En un sistema de tipos estático puro, las definiciones de variables fijan el límite para inferir su tipo; en el mismo sentido, el usar una variable determina el tipo de ésta en un límite superior dependiendo del contexto donde fluya esta variable. De esta forma, el tipo de una variable debe satisfacer ambos conjuntos de restricciones, y asegurar de manera estática que cada aparición de uso de la variable es segura. En contraste, en un sistema de tipos gradual, el uso de una variable no necesariamente es seguro para cada definición de variable: es por ello que los errores en tiempo de ejecución pueden evitarse si cada uso de la variable es seguro para alguna definición. Por lo que el tipo de una variable puede ser inferido considerando sus definiciones desde tiempo de compilación. Las variables de tipo pueden ser inferidas solo si están dentro de un contexto *local*, es decir, si durante tiempo de compilación conocemos todas sus definiciones. Con base en lo anterior, solo se infieren los tipos de aquellas variables para las cuales se conocen todos sus valores en flujo, i.e., sus entradas *(inflows)* y el algoritmo calcula sus tipos de entrada, sin verificar que las soluciones satisfacen la mayor cota inferior del contexto de sus tipos [69].

Los valores de orden superior como son las funciones y los objetos se tratan usando la noción de tipos de tipos, es decir, *kinds*. El algoritmo propuesto por Rastogi et al. garantiza la consistencia del lenguaje: en tiempo de ejecución, si un programa falla usando la inferencia de tipos, éste termina exactamente en el mismo punto que los sistemas de tipos dinámicos. El algoritmo propuesto por Herman et al. comienza la inferencia reemplazando las anotaciones de tipo faltantes en un programa por sus variables de tipo en tiempo de compilación. Una vez que se tienen las variables de tipo el programa genera las coerciones entre tipos. Una coerción es de la forma $S \triangleright T$, donde S y T denota a los tipos. Las coerciones declaran como testigos entre el flujo de un término de tipo S a un contexto de tipo T. Particularmente, las coerciones envuelven variables de tipo que son interpretadas como

[14] Este algoritmo está implementado para el lenguaje de programación ActionScript, el cual es una extensión de JavaScript con sistema de tipos graduales, y está diseñado como un lenguaje de programación que permite ejecutar aplicaciones en Flash [69].

flujos (*flows*) para estas variables de tipo. Por ejemplo, si X denota a una variable de tipo, entonces $T \triangleright X$ hace referencia a un flujo de entada (*inflows*) para X, mientras que $X \triangleright T$ denota a un flujo de salida (*outflow*) para X, y T no es una variable de tipo [69].

Tipificado gradual polimórfico

Los verificadores de lenguajes con tipificado gradual presentan dos componentes principales [51]:

- Tipo dinámico.
- Consistencia de tipos.

El primero de esos componentes regularmente es denotado por $\star$, éste no se encuentra asociado al tipo `any`[15]. La consistencia de tipos es asegurada por el tipificado estático de todas las expresiones, incluyendo el tipo dinámico, es decir, todas las expresiones pueden ser representadas por $\star$ y a su vez los valores pueden ser asociados al tipo $\star$ en tiempo de ejecución. De hecho, la tipificación gradual permite la interacción entre el tipificado estático y el dinámico, incluyendo la interacción de toda la ejecución, por ejemplo, entre $\star \to Int$ y $Int \to \star$ usando la consistencia de tipos. Igarashi et al. proponen un sistema F_G derivado del sistema F propuesto por Girard en 1972 donde se extiende las reglas de TG ahora usando polimorfismo paramétrico. Este sistema F_G clasifica las variables como (1) variables de tipo estático y (2) variables de tipo gradual. La semántica dinámica del sistema F_G está definida por una translación a un nuevo sistema de cálculo de culpa polimórfico F_C. Éste es una extensión de (un subconjunto) del cálculo de culpabilidad de tipos. En el sistema F_G está explícita y completamente tipificado, es decir, cada variable debe de tener un tipo asociado y el tipo abstracción lambda debe ser explícito. Si alguna parte del programa no está tipificada se realiza una inferencia. Debido a lo anterior, en el sistema F_G se puede considerar un lenguaje intermedio después de haber hecho la inferencia de tipos y antes de traducirse al Cálculo de la Culpa (*Blame Calculus*) [51].

La extensión del Cálculo Lambda Sin Tipos (DTLC, del inglés *Untyped Lambda Calculus*) y el Cálculo Lambda con Tipificado Simple (STLC, cuyas siglas significan en inglés *Simply Typed Lambda Calculus*) presentan una extensión conservadora. Esto es que todos los términos sin $\star$ preservan un significado equivalente tanto en el GTLC (*Gradually Typed Lambda Calculus*) como en STLC. Expresado de otro modo, los programas desarrollados en un lenguaje con tipificado gradual no modifican su semántica. Igarashi et al. definen los siguientes conceptos:

- Tipo seguro: todo término bien tipificado en GTLC regresa un valor, o un culpable o bien diverge. Esto asegura que todo término bien tipificado no provoca algún tipo de error.
- Teorema de subtipado-culpa (*Blame-Subtyping Theorem*): este teorema trata acerca de la consistencia de la culpa. Asegura que si hubiera un *cast* entre dos tipos en una relación de subtipificado, (cuya definición es omitida) nunca levanta una culpa. Por otra parte, la relación de subtipificado asegura que al hacer cualquier conversión de tipos, ésta nunca falla, puede ser usada en tiempo de compilación y en algunos casos en tiempo de ejecución para hacer algunas optimizaciones como el reducir o quitar algunas conversiones de tipo redundantes.
- Garantía Gradual (*The Gradual Guarantee*): dentro de un sistema de TG, agregar más tipos estáticos a un programa ayuda a *atrapar* más errores de tipo en tiempo de compilación y tener más culpas en tiempo de ejecución, sin modificar el resultado o comportamiento del programa. Dicho de otro modo, el que un programa presente más anotaciones de tipo estáticas sirve solamente para identificar más errores tanto estáticos como dinámicos sin modificar el comportamiento del programa.

Los contratos manifiestos polimórficos (*Polymorphic Manifest Contracts*) integran el refinamiento de tipos y la verificación dinámica de contratos, garantizando la parametricidad estática de todas las variables de tipo, las cuales deben ser tipificadas estáticamente.

[15] Esto se retomará en la siguiente sección de este trabajo doctoral.

Tipificado gradual abstracto

Se han realizado algunos diseños e investigaciones para extender los lenguajes con tipificado dinámico, con el objetivo que soporten el tipificado gradual. Los diseñadores de los sistemas graduales han aportado herramientas que ayudan a conceptualizar, estructurar, y evaluar sus diseños. El Tipificado Gradual Abstracto, (del inglés *Abstract Gradual Typing* conocido por sus siglas AGT), es el fundamento teórico abstracto de un lenguaje, sustentado por medio de juicios de tipo, basándose en la preexistencia de la TG, dando una interpretación abstracta de éstos; es decir, asignar a los tipos graduales una semántica en términos de los tipos estáticos preexistentes del tipificado gradual, dicho de otro modo presupone que el lenguaje es **consistente**. Garcia y Tanter en [42] expresan que dados los juicios de tipo estáticos con sintaxis dirigida en la AGT tienen su correspondiente juicio de tipo gradual en el tipificado gradual. Este enfoque garantiza que la verificación de tipos en tiempo de ejecución se desarrolla de forma regular. La idea principal es contar con juicios de tipos graduales usando una interpretación abstracta para añadir o agregar un significado (semántica) a los tipos graduales en términos de los tipos estáticos.

El criterio de corrección de lenguajes graduales usados de manera colaborativa

La corrección del tipificado gradual precisa la verificación de tipos en tiempo de ejecución. Mientras más sólida o correcta sea esta verificación, (para el lenguaje objetivo generado), se presentará (a) un menor número de configuraciones que no se puedan seguir evaluando (del inglés *stuck configurations*) o bien (b) se obtendrá un menor número de comportamientos indefinidos. Lo anterior puede causar errores de tipo, difíciles de encontrar y depurar. Ejemplos de lenguajes de programación donde se pueden identificar estas situaciones son Hack y TypeScript, ya que éstos no grantizan la seguridad de tipos en tiempo de ejecución.

El poder combinar verificación de tipos estática y dinámica en el mismo lenguaje es la característica principal del tipificado gradual. En este caso, lenguajes como Dart, Hack, Typed Clojure, TypeScript, Gradualtalk y Reticulated Python, (por mencionar algunos) son ejemplos de lenguajes de programación que presentan este tipo de tipificado. Los programadores que laboran en el sector privado en la generación de software y que utilizan este tipo de lenguajes lo hacen principalmente para desarrollar de manera rápida, prototipos de programas flexibles. Sin embargo, ésta no es su única responsabilidad en el ámbito profesional, muy probablemente también deban seguir dando mantenimiento a los sistemas de software legados o bien realizar migraciones de código legado a lenguajes de programación de creación reciente o de moda. Realizar tales migraciones de un lenguaje a otro resulta una tarea laboriosa y de cuidado, pues se debe preservar la corrección de dichos sistemas al hacer las modificaciones necesarias durante dichas migraciones. Vitousek, Swords y Siek muestran un diseño alternativo de lenguajes con tipificado gradual que satisfaga la corrección del mundo abierto (del inglés *open-world soundness*) presentando una semántica formal. Hacen uso del Teorema de la Culpa (del inglés *Blame Theorem*), para mostrar que el sistema garantice su corrección tanto de manera estática como dinámica. Lo anterior ha provocado un mayor interés en el estudio de lenguajes con tipificado gradual desde las comunidades de investigadores en el área de lenguajes de programación, así como en la industria privada en el área de Tecnologías de la Información y Comunicación, TIC [105].

Dentro del modelo de corrección de mundo abierto, se busca extender la propiedad de corrección en programas cuyos lenguajes incrustados en su código deban ser traducidos a un lenguaje objetivo arbitrario. Vitousek, Swords y Siek proponen un enfoque "con custodias" (del inglés *guarded approach*) en este contexto (corrección de mundo abierto), donde la tipificación gradual falla cuando se genera código a un lenguaje objetivo específico el cual no necesariamente pertenece a la categorización de un lenguaje con tipificado gradual, éste es nombrado por ellos como soporte nativo (del inglés *spartan host*), así un *spartan host* está definido en [105] como:

> "Un lenguaje que carece de soporte nativo en tiempo de ejecución para el tipificado gradual" [16].

Este enfoque se centra del contexto de la garantía gradual, el cual hace referencia al estado de debilitamiento o alteración y en particular quitando anotaciones de tipo de los programas sin introducir errores nuevos a éste. Al estudiar la corrección en este contexto, se debe diferenciar entre los términos del código origen, donde un término e se identifica como $\diamond$, mientras que el contexto del programa C representa el código del lenguaje objetivo, identificado por $\blacklozenge$. Definición de corrección de mundo abierto [105]:

[16] *Spartan host: A language which lacks native runtime support for gradual typing.* [105].

Sea e_s la expresión original con tipo T la cual se traduce de un término e a un lenguaje objetivo. El sistema cumple con la corrección del mundo abierto si, para cualquier contexto C de un programa en el lenguaje objetivo o destino cumple:

- Existe algún valor v tal que $C[e]$ reduce a v en cero o más pasos, o
- $C[e]$ diverge, o
- $C[e]$ se reduce a errores de conversión de tipos en tiempo de ejecución, o
- $C[e]$ se detiene en cero o más pasos con un error, lo cual indica que el término *llega a estancarse o detenerse debido a que está marcado el término como* ⧫.

Este modelo ayuda a los programadores que escriben programas con tipos estáticos a garantizar su corrección, los cuales pueden hacer uso de bibliotecas externas escritas para el lenguaje objetivo usando ahora el módulo para tipos graduales. La corrección de un mundo abierto garantiza que al hacer uso de las bibliotecas de éste, no produce errores de tipo o bien se comporta de manera no deseada.

Tipificado de ocurrencia

El tipificado de ocurrencia (del inglés *occurrence typing*) se refiere a un modelo teórico cuyo proceso principal es la conversión de un programa escrito en un lenguaje con tipificado dinámico a uno con tipos explícitos de manera automática, preservando la propiedad de seguridad descrita por Pierce en [66]. En este tipo de conversión se asume la existencia de un lenguaje tipificado, con la misma semántica que el lenguaje original (sin tipos explícitos), permitiendo que los valores utilizados entre ambos lenguajes se mantengan tanto dentro de los programas escritos en ambos lenguajes, como en los módulos usados en cada uno de ellos. Para probar la factibilidad de este enfoque se implementa el lenguaje Typed Scheme, el cual es una versión del PLT Scheme con tipos explícitos [92]. Dadas las siguientes características del lenguaje resulta una opción adecuada para probar el modelo de ocurrencia de tipos: (1) que se pueden desarrollar programas de *scripting* usados por una comunidad grande de programadores, lo cual conlleva el tener una extensa cantidad de código disponible para probar este modelo y (2) que se tienen mecanismos de extensión del lenguaje como son los macros. La versión tipificada de PLT Scheme combina el enfoque de tipificado de ocurrencia de tipos, junto con el subtipificado, tipos recursivos y polimorfismo, así como la inferencia de tipos. En tal modelo se prueba que el lenguaje mantiene su corrección. Este tipo de lenguajes, (los que permiten tener anotaciones de tipo o no en un programa), demandan reflexionar y estudiar a fondo en los sistemas de tipos. En el siguiente ejemplo se oberva código en Typed Scheme el cual permite tener la definición de un número complejo con dos tipos de constructores, es decir, puede ser tratado como número o como lista de números [92]:

```
1  ;;definición de este daro es un número complejo (Complex) el cual puede ser
2  ;;un número o
3  ;;una lista de números (cons Number Number)
4  ;;Complex > Number
5  (define (creal x)
6  (cond [(number? x) x]
7        [else (car x)]))
```

Código 3.2: Definición de una función para números complejos.

En este código se observa la definición para un número complejo x, el cual puede está representado como un número solamente o bien un par de números por medio de la primitiva `cons`. De manera informal se puede pensar que un número complejo puede representarse tanto como número o un par de números por medio de una lista (usando esta representación), es decir, existe la idea de la existencia de ambos tipos de conjuntos. La función no utiliza una correspondencia de patrones para resolver la unión de conjuntos. Lo único que usa es un predicado que distingue entre ambos casos (`cond`). La primera línea corresponde al código en Typed Scheme presenta dicho predicado:

```
1 (define type alias Cplx (U Number (cons (Number Number)))
2
3 (define: (creal [x:Cplx]): Number
4    (cond [(number? x) x]
```

```
5          [else (car x)]))
```

Código 3.3: Definición de una función para números complejos en Typed Scheme.

En el código anterior se puede apreciar el tipo `Cplx`, el cual es una abreviatura para denotar a los números complejos. Tal abreviación brinda la idea de unión en ambas posibilidades de número complejo. Por otro lado, el cuerpo de la función `creal` se mantiene sin modificaciones tal y como se muestran en el primer caso.

La característica principal de este tipo de lenguaje es poder soportar la asignación de tipos diferentes a diversas ocurrencias de una variable en el transcurso del flujo de un programa. Por ejemplo, la siguiente definición de una función con tipos polimórficos [92]:

$$(\lambda\,(x : \bigcup\ Number\ \ Boolean)\\ \quad (if\ (number?\,x)\ (=\,x\,1)\ (not\,x)))$$

La función anterior entonces recibe la unión de los tipos $Number$ y $Boolean$. Este código regresa un valor de tipo $Boolean$ (es decir, falso o verdadero), por lo que de manera formal es descrita por [92] como:

$$((\bigcup\ Number\ Boolean) \implies Boolean)$$

Si la función anterior recibe un argumento `x` de tipo numérico, ésta se ejecuta de manera correcta, pues tanto en la expresión condicional del `if`, como en la expresión `then` y `else`, la `x` se aplica a cada una de las primitivas de forma adecuada. De esta manera el verificador de tipos no genera ningún tipo de error en el contexto de cada subexpresión del `if`. El verificador de tipos en la rama del `then` asocia en su ambiente el tipo `Number` a la `x`, mientras que en la rama del `else` la `x` es asignada a un tipo `Boolean` [17]. En caso de no recibir un argumento de tipo numérico la función regresará un error, al tratar de evaluarse la expresión condicional del if, pues el predicado $number?$ sólo se aplicará a valores de tipo $number$.

Teoría de Tipos Cuantitativa

Una promesa que ofrece la Teoría de Tipos es que se puede programar y razonar en un mismo sistema. Así la Teoría de Tipos permite tener cierto control sobre ciertos aspectos de un programa: (1) cómo son utilizados los recursos computaciones y (2) cuando pueden ser re-usados. De este modo, el seguimiento de estos recursos por medio de sus tipos, ha sido estudiado usando la lógica lineal de Girard. Actualmente se ha propuesto un sistema que resuelve este tipo de enfoques, utilizando el uso de los tipos (por ejemplo, si no es utilizado, es decir, si éste no se utiliza ni una vez). A este tipo de sistema se le conoce como Teoría de Tipos Cuantitativa (*Quantitative Type Theory*). La Teoría de Tipos expone un enfoque más relajado entre los conceptos de *programación* y *verificación*, combinándolos en un solo sistema. La Teoría de Tipos presenta ciertas dificultades al ser usado como un lenguaje de programación. Sin embargo, ésta usa los datos con dos fines distintos [8]:

- Extensionalmente (*extensionally*): para que la información pueda ser utilizada en la formación de tipos.
- Intensionalmente (*intensionally*): como información computacional la cual es manipulada por los programas.

El objetivo del enfoque de Atkey es proveer una reformularización de un sistema de Tipos Gradual y el sistema de Tipos Cuantitativos para una gramática pequeña dentro de lenguajes tipificados dinámicamente. La Teoría de Tipos Cuantitativa de Martin-Löf, presenta un juicio de términos de la siguiente forma:

$$x_1 : S_1,\, x_2 : S_2,\, \ldots\, , x_n : S_n \vdash M : T$$

En el contexto $x_1 : S_1,\, x_2 : S_2,\, \ldots\, , x_n : S_n$ tiene dos usos. Computacionalmente describe los nombres que se usan para acceder a los recursos que pueden ser utilizados en el término M para poder construir un valor de tipo T. Lógicamente, describe los nombres que se utilizan para referirse a un significado usado para formar los tipos $S_1, S_2, \ldots, S_n$ y T.

[17] Typed Scheme combina la idea de tipificado de ocurrencia, basado en el estudio de los programadores de Scheme, quienes en su mayoría diseñan sus programas basándose en el razonamiento orientada en su flujo con base en los tipos utilizados en éstos [92].

Por otro lado, la Teoría de la lógica lineal de Girard hace uso de los conceptos de ausencia o presencia de variables en un contexto para grabar o recordar qué variables son usadas computacionalmente. Siguiendo el juicio anterior tendremos:

$$x_1 : X_1, \ldots x_n : X_n :\vdash M : Z$$

Este juicio muestra que el término M debe utilizar los recursos computacionales nombrados como $x_1, \ldots, x_n$ una única vez. Lo anterior presenta algunas restricciones en la parte práctica, pues supone que no es necesario un recolector de basura o bien no es necesario preservar estructuras de datos viejas [8].

En este diseño de sistema de tipos se agrega información acerca de las variables tomando en cuenta los contextos:

$$x_1 \overset{\rho_1}{:} S_1, \ldots, x_n \overset{\rho_n}{:} S_n \vdash M : T$$

Donde $\rho_1, \ldots, \rho_n$ son elementos de un semi anillo indicando la correspondencia entre las variables usadas. En este sistema de tipos, el cero es utilizado en un semi anillo, para denotar que la variable no es usada.

Resumen

En este capítulo se presenta una clasificación de los diferentes enfoques relacionados con el tipificado que utilizan algunos lenguajes de programación. Esta clasificación se hace con base en el grado de tipificado que presenten los lenguajes de programación, es decir, dependiendo de qué tan estática o dinámica es su forma de tipificar sus diferentes construcciones. Se describen cada uno de estos enfoques, presentando algunas de sus características principales y lenguajes donde se han aplicado cada uno de los diversos tipificados, así como ejemplos donde se puede observar su aplicación.

A su vez se presentan diferentes enfoques teóricos y prácticos relacionados con la verificación de tipos estática y dinámica, describiendo sus objetivos y características principales: durante la última década, se han realizado una gran cantidad de contribuciones al Tipificado Gradual. Comenzando por el trabajo de [78], muchas investigaciones han estudiado diferentes variantes y temas relacionados con el tipificado gradual a través de diferentes enfoques y aplicaciones tales como: inferencia gradual [41], basada en unificación inferencia [81], tipificado de seguridad (*security-typing*) [33], sumas de refinamiento [52], unión e intersección de tipos [19], uniones recursivas [80], referencias [83], efectos [71, 72], tipificado gradual polimórfico [51], lenguajes Orientados a Objetos [79, 88], *typestates* [43], entre otros.

Capítulo 4

Extensión de un lenguaje con Tipificado Gradual usando Registros

> "Cuando queremos construir, primero examinamos la trama y luego dibujamos el modelo."
>
> *William Shakespeare.*

El presente capítulo expone una reinterpretación de la metodología de Toro y Tanter en la cual se basa esta propuesta de tesis para hacer la extensión de ésta usando una estructura de datos como son los registros. Primero se expone la parte estática de cada una de las transformaciones y posteriormente la dinámica.

Introducción al tipificado gradual

Los lenguajes con tipificado gradual combinan tanto las ventajas presentes en el tipificado estático como en el dinámico [42, 51, 81]. Los lenguajes gradualmente tipificados permiten a los programadores omitir o no utilizar explícitamente los tipos definidos en el lenguaje al programar, garantizando al mismo tiempo la seguridad de los tipos, así como el comportamiento correcto y consistente del programa, tanto en tiempo de compilación como en tiempo de ejecución.

En este trabajo doctoral se centra en los **Tipos de Unión Gradual**, (*Gradual Union Typing, GUT*) propuestos por [95, 96], las cuales son una interpretación de los tipos de unión para tipificado gradual, basados en la metodología de **Tipificado Gradual Abstracto** (*Abstracting Gradual Typing, AGT*) propuesta por [42]. Este último brinda una semántica dinámica directa a un lenguaje gradual a partir de un lenguaje con algunos tipos estáticos.

Los sistemas de tipos utilizados en la interpretación de GUT se extienden desde un sistema de tipos utilizando relaciones de tipo coherentes, lo que permite una combinación del tipo desconocido y las uniones graduales para una verificación de tipo dinámica completa, donde un enfoque estratificado sigue la metodología AGT la cual proporciona una semántica para el tipificado gradual. Ambos enfoques, AGT y GUT, permiten el tipificado gradual y tratan con el concepto de *imprecisión*, ya que los programas pueden presentar información imprecisa de tipos estáticos, que es manejada por el lenguaje de forma tanto estática como dinámicamente, con el fin de ofrecer un lenguaje gradual sólido o consistente.

Por ejemplo, considere la función $\lambda x.x + 1$ cuyo argumento no presenta un tipo asignado de manera explícita, éste será asignado hasta que se pueda aplicar esta abstracción lambda con un argumento real. La imprecisión a nivel de tipos no debe detener la evaluación de una abstracción lambda bien formada, y por lo tanto, un lenguaje gradual puede definir sus tipos graduales en particular porque la abstracción lambda en este tipo de lenguaje

deberá tener asignado un tipo dada su sintaxis $\lambda x : T.x + 1$, es decir T deberá estar definido cada que se defina a una abstracción lambda.

Registros

Los lenguajes de programación incluyen estructuras de datos básicas, donde dos de los tipos de datos más conocidos son los arreglos y los registros. Los arreglos almacenan elementos del mismo tipo, y cuando se selecciona un elemento del arreglo se hace mediante un índice numérico. Por otra parte, los registros almacenan elementos de posiblemente diferentes tipos, accediendo a ellos a través de etiquetas o nombres de campos [38].

Un registro es una estructura de tipos de datos, la cual contiene variables (llamadas campos) y valores asociados a éstas. Los campos pueden tener diferentes tipos. Algunos lenguajes de programación usan registros para así definir nuevos tipos de datos, lo que permite a su vez extender el lenguaje [38].

La implementación de un registro está compuesta por una o más etiquetas. Éstas son denotadas (en este trabajo) por el símbolo ℓ. Cada etiqueta tiene asignado un valor, el cual a su vez tiene asignado un tipo. Cabe resaltar que un registro es mucho más sencillo y eficiente de pasar como argumento de una función, que una colección de valores durante tiempo de ejecución, aún y cuando dicha función se ejecute frecuentemente [38].

Un registro, como ya se ha mencionado, permite tener componentes o elementos de cualquier tipo y que potencialmente pueden ser otros registros. Así, un registro puede utilizarse para representar una estructura similar, pero diferentes objetos de datos. Esta característica permite definir registros con una estructura jerárquica que es consistente aunque tengamos *un registro de registros*. Por ejemplo, consideremos un registro de *employees* (empleados) que incluya los campos *id* (identificador), *name* (nombre), *age* (edad), *salary* (salario), y *type-employee* (tipo de empleado). Una instancia de la misma puede ser un registro llamado A_1 el cual está definido como *[id = 1, name=Foo, age=25, salary=1000, type-employee=A]* donde la etiqueta *id* se refiere al identificador del empleado, *name* a la cadena de nombre correspondiente, *age* al dato numérico referido a su edad, *salary* a la cantidad numérica que recibe por su trabajo y *type-employee* al tipo de empleado (en este caso se define con el tipo A). Se puede pensar que A puede ser también otro tipo de registro donde se definen las (sub)categorías de empleados de la empresa. Por ejemplo, *type-employee* puede representar el tipo de tipo de pago que recibe, por mes o por día. En este caso, el componente de registro tiene dos tipos de variantes en uno de sus elementos [68].

Los registros resultan ser una estructura adecuada para estudiar los beneficios del enfoque gradual de tipos unión. Por otro lado, algunos otros lenguajes presentan estructuras similares en sus núcleos como lo son las estructuras `structs` en Racket [1]. El estudio de registros o una estructura similar es útil, ya que suponen una aproximación a las estructuras de datos para almacenar datos heterogéneos y cuyo acceso a dichos datos se hace mendiante una etiqueta o índice (dependiendo el diseño del lenguaje *per se*).

Por ejemplo, consideremos una función cualquiera f cuyo argumento es un registro y existe una imprecisión dada por la ausencia de información sobre su cuerpo de función, por lo que la firma de f tiene el tipo $[\ell_1 : {T_1}^{1 \epsilon 1..n}] \rightarrow ?$. La aceptación de dicha función durante el tiempo de compilación, y un posible manejo de errores durante el tiempo de ejecución debe soportarse idealmente por un lenguaje con tipificado gradual: dicho lenguaje permite al programador escribir programas dejando algunas anotaciones de tipo sin expresar. Los lenguajes de tipificado gradual que utilizan la interpretación *Gradual Union Typing, GUT*, y por tanto la metodología *Abstracting Gradual Typing, AGT* proporcionan una base general para desarrollar y diseñar la semántica estática y dinámica de los lenguajes que permiten tales comportamientos[2].

Una extensión con registros usando etiquetas

Esta sección describe los conceptos básicos de la Tipificación Gradual, haciendo hincapié en las Uniones Graduales [95], y describiendo el enfoque estratificado de AGT por [96] utilizado en este trabajo (subsección 4.3.3). Con el fin de ofrecer un trabajo autocontenido, en los siguientes subapartados se presenta una introducción al

[1] Los *structs* contienen un nombre, campos y funciones selectoras [35].
[2] Tanto la GUT como la AGT se describen en las siguientes subsecciones de este trabajo.

Tipificado Gradual Abstracto (*Abstracting Gradual Typing, AGT*) donde se explica la metodología para derivar un lenguaje de tipo gradual 4.3.1 seguido de una subsección 4.3.3 para explicar brevemente las uniones graduales.

Al explorar la idea de imprecisión en los sistemas de tipos, es decir, cuando un término puede tener cualquier tipo estáticamente, un posible enfoque es considerar el tipo `any` como un supertipo [3] [92] para asignar un tipo a un término durante el tiempo de compilación. Sin embargo, esta propuesta puede ser engañosa, ya que la imprecisión puede presentarse como que tiene menos información acerca de sus tipos, y no necesariamente la ausencia de información.

Para reducir la imprecisión, se explora otro enfoque en los lenguajes con tipificado híbrido, donde un tipo no puede ser determinado en tiempo de compilación [1]. En tales lenguajes, los programadores tienen que insertar coerciones hacia y desde, el tipo `Dinámico` (*Dynamic*) [4] [37]. En algunos lenguajes de tipificación dinámica el tipo *Dynamic* es declarado como supertipo de todos los tipos [46].

Sin embargo, cuando se relajan las asignaciones de tipo para que se completen durante tiempo de ejecución, la imprecisión de tipos se interpreta como la imposibilidad de asignar un tipo a un término de forma estática, para así dar un tipo desconocido que debe ser resuelto en tiempo de ejecución.

Un lenguaje de tipificación gradual añade el tipo desconocido, denotado en estos lenguajes como ?, para enfatizar la imprecisión [42, 78]. En este contexto, es importante aclarar que el supertipo `any` no coincide con el tipo desconocido, ya que un valor del tipo `any` no puede ser utilizado como argumento de ninguna función, es decir, `T <: any`, pero no necesariamente `any <: T` [66]. Esta propiedad está permitida en los lenguajes tipificados gradualmente como se detalla más adelante.

El uso del tipo desconocido ? está restringido, es decir, el programador puede o no tipificar su código; sin embargo, el lenguaje a través del verificador de tipos debe añadir anotaciones de tipo asignando el tipo ?, para permitir una verificación estática y preservar la consistencia del programa durante tiempo de ejecución. El tipo desconocido se utiliza en el lenguaje para representar todos los tipos posibles que no están explícitamente escritos por el programador. Es importante recalcar que el tipo desconocido no está disponible para el programador en ningún momento. Por lo que, se puede generar una imprecisión de tipo, debido a las anotaciones de tipo que faltan y de este modo el lenguaje deberá detectar errores de tipificación. Esto se debe de garantizar mediante una relación de consistencia (denotada como $T \sim T'$) la cual debe mantenerse durante el tiempo de ejecución.

Las relaciones de consistencia son un componente importante en cualquier lenguaje con tipificado gradual. En particular, la relación de consistencia es la contrapartida gradual de la igualdad. Dicha relación es necesaria para garantizar que los programas no fallen, es decir, que no devuelvan un tipo inseguro. La relación de consistencia de tipo $T \sim T'$ se define cuando dos tipos están relacionados durante el tiempo de ejecución, en presencia del tipo ? [78]:

$$\frac{}{T \sim T} \qquad \frac{}{T \sim ?} \qquad \frac{}{? \sim T} \qquad \frac{T_{11} \sim T_{21} \qquad T_{12} \sim T_{22}}{T_{11} \to T_{12} \sim T_{21} \to T_{22}}$$

Es importante destacar que un verificador de tipos de un lenguaje con tipificado gradual debe identificar posibles errores de tipo antes de la ejecución, garantizando la consistencia de tipos, tanto en tiempo de compilación como en tiempo de ejecución. La detección puede llevarse a cabo durante la verificación de tipos. La traducción al Cálculo de Conversión de Tipos (*Cast Calculus*) [78] se proporciona para garantizar que no se violen los supuestos estáticos en tiempo de ejecución, insertando la conversión explícita de tipos verificando los límites entre los tipos con diferente precisión [95].

Por ejemplo, en lenguajes con tipificado dinámicos (como Racket, Scheme o Lisp), supóngase una función definida como `(lambda (x) if x then (+ x 1) else (not x))`. En esta declaración, sólo se indica que el tipo de `x` es desconocido en este momento. Más tarde, durante tiempo de ejecución, el tipo de `x` debe ser asignado con un valor, y debido a este valor (`true` o `false`) el tipo del programa puede ser un número (si `x` se reduce a verdadero) o un booleano (si el valor de `x` fuera `false`).

La inclusión del tipo desconocido en un sistema de tipos permite la aceptación de más programas en tiempo de compilación que un sistema que no considera una comprobación dinámica de tipos, como en el ejemplo anterior.

[3] También conocido como top ($\top$).

[4] Otros autores utilizan el símbolo $\star$ para denotar un tipo dinámico [51, 69].

Algunos otros programas pueden llevar a una ejecución errónea y deben ser rechazados o al menos advertir al programador de un mal comportamiento. La GUT combina el tipo desconocido y las uniones graduales para añadir flexibilidad en un lenguaje para manejar este tipo de programas [95].

Tipificado Gradual Abstracto

La interpretación abstracta de la tipificación gradual se define como "dar a los tipos graduales una semántica en términos de tipos estáticos preexistentes" por [42]. El Tipificado Gradual Abstracto (*Abstracting Gradual Typing, AGT*) induce una semántica dinámica para programas con tipificado graduales, dada una sintaxis dirigida por un juicio de tipo estático en su correspondiente juicio de tipo gradual.

El AGT permite juicios de tipificación graduales utilizando predicados consistentes basados en la interpretación abstracta, para dar una semántica a los tipos graduales en términos de tipos estáticos. La idea es pensar en un tipo gradual como un conjunto de tipos estáticos. En este caso, los predicados pueden levantarse [5] para aplicarse a los tipos graduales.

En [12] se sintetiza una gran cantidad de trabajo relacionado sobre la tipificación gradual y el uso de la interpretación abstracta, la cual se utiliza como base para el desarrollo de [42]. Así, la AGT brinda una semántica estática y dinámica para un lenguaje que satisface todos los criterios semánticos dados en un lenguaje de tipificación gradual por *construcción*, definiendo un método incremental [42]. Este enfoque satisface los criterios refinados de los lenguajes de tipificación gradual establecidos por [82].

El trabajo presentado aquí se realiza por construcción partiendo de un Lenguaje Funcional de Tipos Simples (STFL) como primer lenguaje [42] que incluye tipos primitivos, en este caso enteros y booleanos. A continuación, se realiza un levantamiento de tipos para obtener un Lenguaje Funcional de Tipificado Gradual (GTFL) el cual extiende el lenguaje con el tipo desconocido, donde es necesario probar nociones como la consistencia gradual y la precisión de los tipos.

La definición de consistencia en este enfoque es la misma que se define en la tipificación gradual, mientras que la precisión de tipos relaciona dos tipos mediante una función de **concretización** γ, tomando un tipo gradual para obtener el conjunto de posibles tipos (estáticos) de un término.

Un tipo T_1 es menos impreciso (o más preciso) que T_2, denotado por $T_1 \sqsubseteq T_2$, si y sólo si $\gamma(T_1) \subseteq \gamma(T_2)$.

También se necesita una función de **abstracción** para refinar todos los tipos posibles de un término en tipos estáticos.

Se debe comprobar que la extensión gradual GTFL es segura, es decir, se debe demostrar que se mantienen (o preservan) las propiedades estáticas y dinámicas como la consistencia, la precisión de los términos y los tipos, junto con las nociones de preservación y progreso en el marco del enfoque gradual.

Unión Gradual

Existen dos enfoques relacionados con la unión de tipos, los cuales han sido analizados por [95]:

1. Uniones etiquetadas o uniones disjuntas de tipos: en este tipo de uniones los tipos deben estar explícitamente etiquetados, por lo que cada elemento tiene un tipo único. Se suelen denotar por $T_1 + T_2$.
   La idea detrás de este enfoque es que las sumas tienen todas las variantes que puede manejar el lenguaje, pero sin introducir errores de tipo. La no-ambigüedad explícita de este tipo de uniones está soportada mediante el cazamiento de patrones [66].

2. Uniones no etiquetadas: donde $T_1 \vee T_2$ denota un valor que puede tener el tipo T_1 o T_2, sin utilizar ningún tipo de mecanismo de etiquetado para saber si presenta ambigüedad o no.
   Por ejemplo, la función $\lambda x : \mathit{Bool}.\, \mathit{if}\ x\ \mathit{then}\ 1\ \mathit{else}\ \mathit{false}$ tiene el tipo $\mathit{Bool} \to \mathit{Int} \vee \mathit{Bool}$, es decir, la función recibe un valor de tipo *Bool* y devuelve un *Int* o un *Bool* dependiendo del valor del argumento que se utilice en el cuerpo de la función. Por lo tanto, un lenguaje con uniones no etiquetadas puede aceptar expresiones que no están bien tipificadas, y en este caso, no hay manera de manejar la ambigüedad.

[5] De la traducción de *predicates can be lifting*.

La combinación de ambas, uniones etiquetadas y no etiquetadas, da un *tipo gradual*, denotado por $T_1 \oplus T_2$, como se propone en [95], el cual abstrae ambos, es decir, tanto a T_1 como a T_2. La idea se basa en que la metodología AGT, la cual permite combinar la unión de tipos y los tipos graduales. Por lo tanto, los tipos de unión gradual ($T1 \oplus T2$) combina ambos enfoques.

Se analiza el siguiente ejemplo: considérese una función f dada por $\lambda x : Int \oplus Bool.x + 1$. Bajo el enfoque de unión gradual, x puede tener un tipo entero (*Int*) o un tipo booleano (*Bool*). Al aplicar un entero a la función, como (f 1), la variable x se sustituye por un entero, y por lo tanto, el cuerpo de la función se evalúa con el valor 2, cuyo tipo es *Int*. Sin embargo, si la función f es llamada con un valor booleano, como (f *true*), diremos que la aplicación está bien tipificada debido a la inserción de la unión gradual, y por lo tanto, la evaluación es segura ya que la violación de los tipos es capturada por un error en tiempo de ejecución.

La unión gradual garantiza una forma segura de tratar estáticamente la imprecisión de un tipo gradual: da tipo a un valor sugiriendo que puede tener ambos tipos, a la vez que restringe la imprecisión haciendo uso de los valores con tipos estáticos. Este enfoque es optimista porque algunas expresiones que en principio deben pasar la verificación de tipos en tiempo de compilación, no son rechazadas por el verificador de tipos. Hasta el tiempo de ejecución, esas expresiones son verificadas y posiblemente rechazadas [95].

Las uniones graduales para la Tipificación Gradual utilizadas aquí, tal y como propone [95], se derivan utilizando un enfoque estratificado de AGT a través de una combinación del enfoque clásico de conjuntos y de las interpretaciones de unión (*union interpretations*), como se detalla en la siguiente sección.

Metodología de Tipificado de Uniones Graduales

La metodología utilizada en este trabajo es conocida como Tipificado de Uniones Graduales, (se traduce del inglés *Gradual Union Typing, GUT*). La unión gradual se expresa como $T_1 \oplus T_2$ para combinar las ideas principales de los enfoques de los tipos de unión, presentados anteriormente. Así, $T_1 \oplus T_2$ representa un tipo gradual, que abstrae T_1 y T_2, y utilizando el enfoque estratificado de AGT [42, 95], para derivar un lenguaje con tipificado gradual, combinando uniones graduales y el tipo desconocido.

La Figura 4.1 muestra las traducciones entre lenguajes (véase la parte superior) entre un Lenguaje Funcional con Tipos Simple (**STFL**) (el óvalo verde en la parte superior izquierda de la figura) y un Lenguaje Funcional con Tipos Graduales (**GTFL**) (el óvalo amarillo en la parte superior central de la figura) y el último óvalo representa la traducción de un lenguaje funcional de tipo gradual el cual incluye el tipo desconocido, el Lenguaje con Tipos de Unión Gradual (representado por el óvalo azul en la parte superior derecha). Estas relaciones están consolidadas semánticamente por los tipos, las cuales en cada lenguaje, utilizan sus respectivas funciones de abstracción y concretización entre sus categorías sintácticas de tipos. Es importante mencionar que, dado que estamos utilizando la metodología AGT, el significado del tipo gradual pertenece al conjunto de tipos estáticos que posiblemente representa, por medio de la función de concretización (γ). Por otro lado la función de abstracción (α) se define como una función de abstracción óptima debido al significado que tiene dentro de la metodología AGT y también, debido a las conexiones de Galois, la función puede utilizarse para levantar predicados de tipos estáticos y las funciones que operan con estos tipos [42, 95].

La traducción de un lenguaje totalmente estático a un lenguaje con tipificado gradual es la siguiente:

- La primera traducción extiende un sistema de tipos estáticos a un sistema de tipos graduales mediante las funciones $\alpha_?$ y $\gamma_?$.
- La siguiente conversión es un levantamiento finito del $P(P(TYPE))$, y $P(GTYPE)$ para obtener una interpretación clásica del conjunto de tipos graduales con las funciones $\widehat{\alpha_?}$ y $\widehat{\gamma_?}$.
- Y finalmente, una tercera traducción incorpora la unión gradual, utilizando las funciones $\alpha_\oplus$ y $\gamma_\oplus$.

En el centro de la Figura 4.1, se muestra el levantamiento entre conjuntos de tipos (generado por la función de conjunto potencia) de $P(TYPE)$ y $GTYPE$, utilizando las mismas funciones de abstracción y concretización $\alpha_?$ y $\gamma_?$.

Este levantamiento se conoce como *interpretación clásica*, aquí a través de una conexión de Galois definida por las funciones de abstracción y concretización. Este tipo de interpretación ya admite el tipo desconocido: **?**

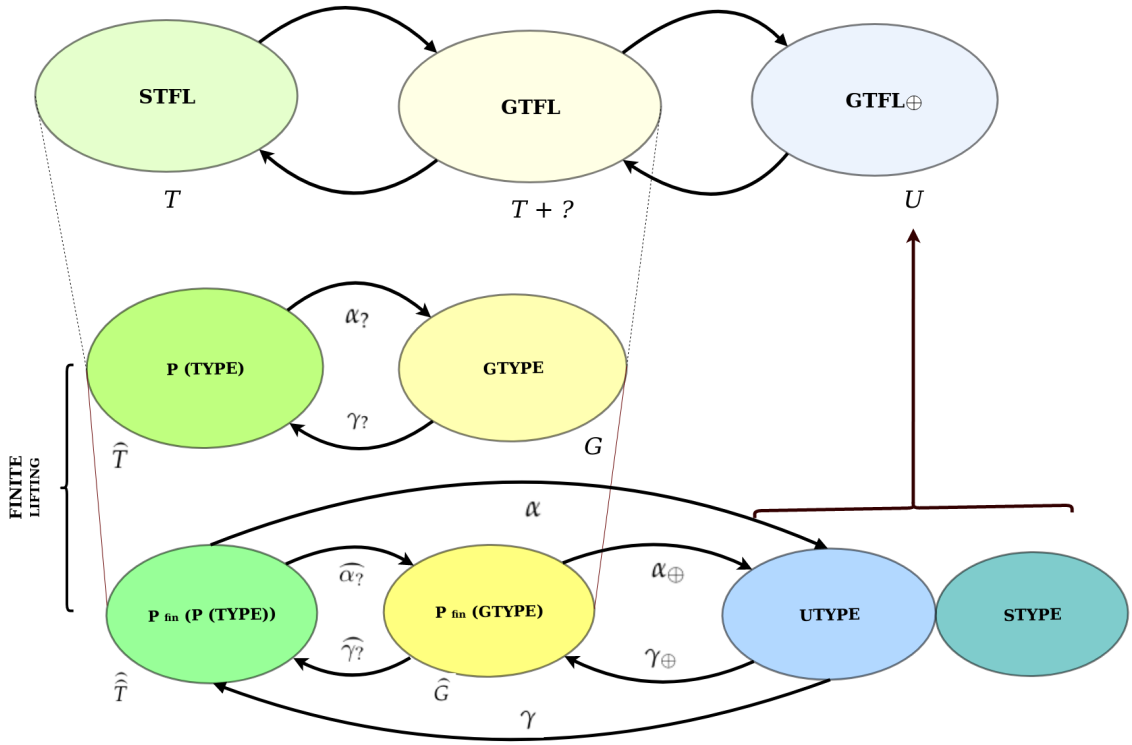

Figura 4.1: Descripción de las categorías sintácticas para tipos bajo la Interpretación de Uniones Graduales.

En la parte inferior de la Figura 4.1, y con el fin de manejar conjuntos finitos de tipos graduales, se necesita un levantamiento finito. Esto forma una nueva conexión de Galois entre lenguajes, detallada para conjuntos de tipos $P\left(P\left(TYPE\right)\right)$ y $P\left(GTYPE\right)$, utilizando la función de abstracción $(\widehat{\alpha_?})$ y una función de concretización $(\widehat{\gamma_?})$. Esto se conoce como *interpretación clásica de conjuntos*. Es importante mencionar que γ es la función de concretización entre $P\left(P\left(TYPE\right)\right)$ y $UTYPE$; y α es la función de abstracción entre $UTYPE$ y $P\left(P\left(TYPE\right)\right)$. Por último, para obtener un lenguaje gradual que combina tanto el tipo gradual tradicional ? como las uniones graduales, la función de abstracción $\alpha_\oplus$ y una función de concretización entre $P\left(GTYPE\right)$ y $UTYPE$.

La última transformación es una combinación entre la interpretación unión y la interpretación clásica de conjuntos, obteniendo la **interpretación estratificada** de tipos graduales [42]. En este lenguaje, los tipos se dividen en dos categorías: $UTYPE$ y $STYPE$. Donde $STYPE$ incluye los tipos graduales formados únicamente por uniones graduales, es decir, uniones graduales sin el tipo desconocido. Por su parte, los tipos graduales en $GTFL^\oplus$ combinan ambos constructores (uniones graduales y el tipo desconocido), denotado por $UTYPE$ [96].

Tipificado Gradual tipo Unión

Un Lenguaje Funcional de Tipos Simples, STFL_{rec} que incluye enteros y booleanos como tipos primitivos, es el lenguaje estático, el cual es el sistema base para desarrollar el sistema de Tipificación Gradual. El sistema de tipos incluye registros como estructura de datos para describir variables (llamadas campos) y sus valores.

Un registro está compuesto por una o más etiquetas, aquí denotadas como ℓ, que pertenece a un conjunto predeterminado $LABEL$. Por lo que, cada campo tiene una etiqueta que tiene un término asignado, que también tiene un tipo asociado. Los campos pueden tener diferentes tipos. Algunos lenguajes de programación utilizan registros para definir nuevos tipos, y por lo tanto, permiten ampliar fácilmente un lenguaje determinado [38, 66]: el agregar registros permite mantener información no homogénea en una misma estructura, y tener la posibilidad

de acceder a través de un identificador de cada uno de sus campos, asimismo añadir registros permite extender un lenguaje manteniendo información compleja en la misma estructura, teniendo la posibilidad de acceder a través de un identificador a cada uno de sus campos (Véase la Figura 4.2).

$$T \in TYPE, \quad x \in VAR, \quad t \in TERM, \quad \Gamma \in VAR \xrightarrow{\text{fin}} TYPE, \quad \ell \in LABEL$$

$$\begin{array}{lcll} \Gamma & ::= & \cdot \mid \Gamma, x:T & \text{(contextos)} \\ T & ::= & Int \mid Bool \mid T \to T \mid [\ell_i : T_i{}^{i\in 1..n}] & \text{(tipos)} \\ v & ::= & n \mid true \mid false \mid \lambda x:T.\, t \mid v :: T & \\ & \mid & [\ell_i = v_i{}^{i\in 1..n}] & \text{(valores)} \\ t & ::= & v \mid x \mid t + t \mid if\ t\ then\ t\ else\ t \mid t :: T \mid t\, t & \\ & \mid & [\ell_i = t_i{}^{i\in 1..n}] \mid t.\ell_j & \text{(términos)} \end{array}$$

$$\frac{}{\Gamma \vdash n : Int}\ (T_n) \qquad \frac{}{\Gamma \vdash b : Bool}\ (T_b) \qquad \frac{x:T \in \Gamma}{\Gamma \vdash x : T}\ (T_x)$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad T_1 = Int \qquad \Gamma \vdash t_2 : T_2 \qquad T_2 = Int}{\Gamma \vdash t_1 + t_2 : Int}\ (T_+)$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad T_1 = Bool \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3}{\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3 : equate\,(T_2, T_3)}\ (T_{if}) \qquad \frac{\Gamma \vdash t : T \qquad T = T_1}{\Gamma \vdash t :: T_1 : T_1}\ (T_{::})$$

$$\frac{\Gamma, x:T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x:T_1.\, t) : T_1 \to T_2}\ (T_\lambda) \qquad \frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_2 = dom\,(T_1)}{\Gamma \vdash t_1\, t_2 : cod\,(T_1)}\ (T_{app})$$

$$\frac{\Gamma \vdash t_i : T_i \text{ para cada } i}{\Gamma \vdash [\ell_i = t_i{}^{i\in 1..n}] : [\ell_i : T_i{}^{i\in 1..n}]}\ (T_{rec}) \qquad \frac{\Gamma \vdash t : T \qquad T = [\ell_i : T_i{}^{i\in 1..n}]}{\Gamma \vdash t.\ell_j : proj(T, \ell_j)}\ (T_{proj})$$

$$\begin{array}{c} dom : TYPE \to TYPE \\ dom\,(T_1 \to T_2) = T_1 \\ dom\,(T) \text{ indefinido en otro caso} \end{array} \qquad \begin{array}{c} cod : TYPE \to TYPE \\ cod\,(T_1 \to T_2) = T_2 \\ cod\,(T) \text{ indefinido en otro caso} \end{array}$$

$$\begin{array}{c} equate : TYPE \times TYPE \to TYPE \\ equate\,(T, T) = T \\ equate\,(T_1, T_2) \text{ indefinido en otro caso} \end{array} \qquad \begin{array}{c} proj : TYPE \times LABEL \to TYPE \\ proj([\ell_i : T_i{}^{i\in 1..n}], \ell_j) = T_j \text{ para alguna } j \in 1..n \\ proj(T, \ell_j) \text{ indefinido en otro caso} \end{array}$$

Figura 4.2: Sintaxis y Sistema de Tipos del (STFL_{rec}) [42] extendido con registros.

La Figura 4.2 muestra la extensión basada en el sistema de [95]. Es importante notar que $\cdot$ denota un contexto vacío y Γ denota un contexto no vacío. Los tipos T se definen como enteros (Int), booleanos ($Bool$), tipo función (también conocido como el tipo flecha), y el tipo registro $[\ell_i : T_i{}^{i\in 1..n}]$. Los valores v son denotadas por las constantes enteras n y las booleanas b [6], las abstracciones λ tipificadas, la adscripción de un valor asociado a su tipo, y los registros $[\ell_i = v_i{}^{i\in 1..n}]$. Por último se presentan los términos, que son cualquier valor v, variables x, sumas de dos términos $t\ +\ t$, condicional $if\ t\ then\ t\ else\ t$; la adscripción $t :: T$; la aplicación de función $t\ t$; el término para registros $[\ell_i = t_i{}^{i\in 1..n}]$; y la proyección sobre un registro $t.\ell_j$.

En este trabajo se agrega la notación para registros: un registro de longitud n se denota por $[\ell_i : T_i{}^{i\in 1..n}]$, con etiquetas ℓ_i. Un término registro se expresa como $[\ell_i = t_i{}^{i\in 1..n}]$, donde ℓ_i denota un campo para el término t_i y el valor de un registro se define como $[\ell_i = v_i{}^{i\in 1..n}]$, donde cada v_i es un valor (hay que tener en cuenta que cada registro tiene un tamaño fijo).

[6] Las metavariables n y b denotan instancias de constantes enteras y booleanas respectivamente.

Las reglas de tipificación en STFL son las usuales, la regla para las constantes enteras (T_n) no requieren ninguna premisa en el antecedente, al igual que la regla para las constantes booleanas (T_b). La regla para una variable tiene asociado un tipo, asumiendo que lo tienen (T_x) la premisa de que $x : T \in \Gamma$ se lee como: "el tipo que se asume para x en Γ es T". La regla para las sumas (T_+) asume que en el contexto Γ el término t_1 tiene un tipo T_1 y el término t_2 un tipo T_2 por lo que ambos tipos T_1 y T_2 deben de ser tipos enteros Int, así al evaluar la expresión $t_1 + t_2$ el valor de esta expresión debe de ser tipo entero bajo el mismo contexto. La regla de tipificación para la expresión (T_{if}) debe de cumplir que en el contexto Γ si el término t_1 al evaluarse tiene asociado el tipo T_1 este T_1 debe ser de tipo booleano, el término t_2 debe ser de tipo T_2 y el tipo del término t_3 debe ser de tipo T_3, así al evaluarse la expresión condicional del if si ésta resulta en un valor de verdadero entonces el tipo de regreso de esa expresión será T_2 o si es falso será de tipo T_3 y ambos tipos T_2 y T_3 deben de ser del mismo tipo. La regla de tipificación para la adscripción ($T_::$), asume que en el contexto Γ el término t tiene asociado el tipo T, T es exactamente el mismo tipo T_1, por lo que en el mismo contexto Γ el término t se encuentra adscrito al tipo T_1, que es justamente el tipo T_1. La regla de tipificación para la abstracción-λ, (T_λ), asume que dado el contexto Γ con la variable x ligada a su tipo T_1 el tipo del término t debe ser T_2, así el tipo de la abstracción-λ es el tipo de su argumento T_1 y regresa el tipo T_2 representado con la notación de flecha: $T_1 \to T_2$. La aplicación de función (T_{app}), donde si t_1 se evalúa a un tipo T_1 y t_2 a un tipo T_2 (asumiendo que los valores representados por las variables libres tienen los tipos asumidos por éstas en Γ) y $T_2 = dom(T_1)$ entonces el resultado de aplicar t_1 a t_2 será un valor con tipo del codominio de T_1. Las funciones parciales de dominio (dom) y codominio (cod) abstraen cada una este tipo de información de una función, la función ($equate$) verifica que los tipos de las ramas de la condicional sean del mismo tipo, y la función parcial $proj$ que recibe un tipo registro y una etiqueta y ésta regresa el tipo asociado a la $j$ésima etiqueta en la proyección. Estas cuatro funciones parciales se usan explícitamente en lugar de depender de las coincidencias de las metavariables.

Las reglas de tipificación para registros son T_{rec} y T_{proj}. Están inspiradas en las reglas de tipificación que aparecen en [66]:

(T_{rec}) Para asignar un tipo a un término de registro, si cada término t_i tiene el tipo T_i bajo Γ, entonces el término $[\ell_i = t_i{}^{i \in 1..n}]$ tiene el tipo $[\ell_i : T_i{}^{i \in 1..n}]$ en el mismo contexto Γ.

(T_{proj}) Para asignar un tipo a una proyección j de un término t en el contexto Γ, el término debe tener un tipo de registro $[\ell_i : T_i{}^{i \in 1..n}]$, y así, un término t_j es la proyección obtenida a través de la etiqueta ℓ_j, y tiene el tipo T_j. Esto viene determinado por la función $proj$.

En este trabajo, se supone que un registro está formado por campos y sus términos asociados, donde cada etiqueta mantiene un orden en los tipos de datos del registro. Debido al diseño del lenguaje que se ha expuesto aquí, vamos a suponer que dos o más registros son diferentes si no tienen exactamente el mismo orden y estructura en la composición de sus campos. Por ejemplo considérese dos registros con las siguientes formas: $[\ell_1 : 49, \ell_2 : true]$ y $[\ell_2 : true, \ell_1 : 49]$. Se consideran diferentes, aunque tienen las mismas etiquetas y valores asociados a sus tipos (respectivamente).

Para este trabajo se presenta un diseño del lenguaje donde dos o más registros pueden ser comparados sólo si tienen el mismo número de campos, y las mismas etiquetas, que deberán estar a su vez en el mismo orden. Esto es lo que llamamos extensión conservadora del lenguaje del Cálculo Lambda con Tipos Simples [7] con registros (STFL_{rec}). Es importante notar que ésta es una elección de diseño particular de un lenguaje, lo que significa que puede ser diferente en otros lenguajes con respecto a su propio diseño [25, 42, 66, 69, 79, 107].

Los registros se pueden implementar de diferentes formas, lo cual dependerá del diseño del lenguaje de programación. Por ejemplo, los registros se representan como tuplas n-arias, compuestas por una etiqueta y su valor respectivo [66].

A continuación se presentan algunos ejemplos de derivaciones de tipos usando el sistema STFL_{rec} definido en la Figura 4.2. El primer ejemplo muestra el uso de la derivación de una expresión de suma binaria, cuyo lado izquierdo es el número entero 2 y el lado derecho el número entero 3, por lo que se cumple la regla de tipificación T_+ ya que ambos lados de la suma son de tipo entero (Int). El segundo ejemplo muestra una expresión condicioonal if donde la condición de ésta es la constante booleana true, la expresión asociada a la rama del then es el valor entero 1 y la expresión asociada a la rama del else es el valor entero 2, por lo que al aplicar

[7] Del inglés *Simply Typed Lambda Calculus*

la regla (T_{if}) se cumple que la expresión true es de tipo booleano, pues de hecho es la constante booleana verdadera, y al usar la función equate entre el tipo del valor 1 y 2 se mantienen asociados al mismo tipo (Int) por lo que la expresión está bien construida y los tipos son adecuados en cada una de sus subexpresiones. El tercer ejemplo, muestra la aplicación de función formada por dos expresiones, la primera es una abstracción lambda y el argumento de ésta es el entero 2. Por un lado la abstracción lambda se expresa como una función cuyo argumento es el identificador x de tipo entero y cuyo cuerpo de función es la proyección de una registro con dos elementos, descritos con las etiquetas ℓ_1 y ℓ_2. Y en el cuarto ejemplo, se encuentra una aplicación de función, donde la primera expresión es una abstracción lambda y el segundo es el identificador y el cual no está tipificado explícitamente. El primer argumento de la aplicación de función es una abstracción lambda cuyo argumento es el identificador x de tipo entero y el cuerpo de esa función es la proyección de un registro, éste a su vez está construido con dos elementos, cuyas etiquetas son ℓ_1 y ℓ_2 con valores 1 y *true* respectivamente. Cabe mencionar que en los ejemplos se muestran de manera explícita las evaluaciones de las funciones *dom*, *cod*, *equate* y *proj*.

Primer ejemplo usando (T_+)

$$\frac{\dfrac{}{\Gamma \vdash 2 : Int}\,(T_n) \qquad Int = Int \qquad \dfrac{}{\Gamma \vdash 3 : Int}\,(T_n) \qquad Int = Int}{\Gamma \vdash 2 + 3 : Int}\,(T_+)$$

Segundo ejemplo usando (T_{if})

$$\frac{\dfrac{}{\Gamma \vdash true : Bool}\,(T_b) \qquad Bool = Bool \qquad \dfrac{}{\Gamma \vdash 1 : Int}\,(T_n) \qquad \dfrac{}{\Gamma \vdash 2 : Int}\,(T_n)}{\Gamma \vdash if\ true\ then\ 1\ else\ 2 : equate(Int, Int)}\,(T_{if})$$

Tercer ejemplo usando (T_{app}, T_λ, T_b, T_{proj}, T_{rec}, T_n y T_b)

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : Int \vdash 1 : Int}\,(T_n) \quad \dfrac{}{\Gamma, x : Int \vdash true : Bool}\,(T_b)}{\Gamma, x : Int \vdash [\ell_1 = 1, \ell_2 = true] : [\ell_1 : Int, \ell_2 : Bool]}\,(T_{rec})}{\Gamma, x : Int \vdash [\ell_1 = 1, \ell_2 = true].\ell_2 : Bool}\,(T_{proj})}{\Gamma \vdash \lambda x : Int.\,([\ell_1 = 1, \ell_2 = true].\ell_2) : Int \to Bool}\,(T_\lambda) \quad \dfrac{}{\Gamma \vdash 2 : Int}\,(T_n) \qquad Int = Int}{\Gamma \vdash ((\lambda x : Int.\,([\ell_1 = 1, \ell_2 = true].\ell_2))\ 2) : Bool}\,(T_{app})$$

Cuarto ejemplo usando (T_{app}, T_λ, T_x, T_{proj}, T_{rec}, T_n y T_b)

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : Int \vdash 1 : Int}\,(T_n) \quad \dfrac{}{\Gamma, x : Int \vdash true : Bool}\,(T_b)}{\Gamma, x : Int \vdash [\ell_1 = 1, \ell_2 = true] : [\ell_1 : Int, \ell_2 : Bool]}\,(T_{rec})}{\Gamma, x : Int \vdash [\ell_1 = 1, \ell_2 = true].\ell_2 : Bool}\,(T_{proj})}{\Gamma \vdash \lambda x : Int.\,([\ell_1 = 1, \ell_2 = true].\ell_2) : Int \to Bool}\,(T_\lambda) \quad \dfrac{y : T \in \Gamma}{\Gamma \vdash y : T}\,(T_x) \qquad T = Int}{\Gamma \vdash ((\lambda x : Int.\,([\ell_1 = 1, \ell_2 = true].\ell_2))\ y) : Bool}\,(T_{app})$$

Figura 4.3: Ejemplos de derivaciones de tipos en STFL_{rec}.

Para probar la propiedad de seguridad de tipos en STFL, es necesario primero dar las demostraciones para (1) el Lema de Sustitución, (2) la Proposición → está bien definido (ver Proposición 4.3.1), (3) los marcos de tipificado [8], y (4) la Proposición ↦ está bien definida[9]. Todas las pruebas se realizan por inducción estructural, y en este trabajo se proporcionan las demostraciones para los casos con registros.

Para realizar la demostración del Lema de Sustitución primero se requiere proporcionar la siguiente definición para registros.

Definición 4.3.1 (Sustitución)**.**

Registros:

$$[\ell_1 = t_1, \ell_2 = t_2, \ldots, \ell_n = t_n][v/x] \;=\; [\ell_1 = t_1[v/x], \ell_2 = t_2[v/x], \ldots, \ell_n = t_n[v/x]]$$

Proyección de un registro: $(t.\ell_j)\,[v/x] = (t[v/x].\ell_j)$

Los siguientes dos lemas (4.3.1 y 4.3.2) se extienden para registros. El Lema de Inversión para la relación de tipificado (ver Lema 4.3.1) se extiende de [95] para registros.

Lema 4.3.1 (Lema de Inversión para la Relación de Tipificado)**.**

1. *Si* $\Gamma \vdash true : T$*, entonces* $T = Bool$*.*
2. *Si* $\Gamma \vdash false : T$*, entonces* $T = Bool$*.*
3. *Si* $\Gamma \vdash n : T$*, entonces* $T = Int$*.*
4. *Si* $\Gamma \vdash x : T$*, entonces* $x : T \in \Gamma$*.*
5. *Si* $\Gamma \vdash t_1 + t_2 : T$*, entonces* $\Gamma \vdash t_1 : T_1$*,* $\Gamma \vdash t_2 : T_2$ *y* $T = T_1 = T_2 = Int$*.*
6. *Si* $\Gamma \vdash$ *if* t_1 *then* t_2 *else* $t_3 : T$*, entonces* $\Gamma \vdash t_1 : Bool$*,* $\Gamma \vdash t_2 : T$ *y* $\Gamma \vdash t_3 : T$*.*
7. *Si* $\Gamma \vdash \lambda x : T_1.\, t_2 : T$*, entonces* $T = T_1 \to T_2$ *para algún* T_2 *con* $\Gamma, x : T_1 \vdash t_2 : T_2$*.*
8. *Si* $\Gamma \vdash t_1\, t_2 : T$*, entonces existe algún tipo* T_{11} *tal que* $\Gamma \vdash t_1 : T_{11} \to T$ *y* $\Gamma \vdash t_2 : T_{11}$*.*
9. *Si* $\Gamma \vdash [\ell_i = {t_i}^{i \in 1..n}] : T$*, entonces* $T = [\ell_i : {T_i}^{i \in 1..n}]$ *donde* $\Gamma \vdash t_i : T_i$ *para cada* $i \in 1..n$*.*
10. *Si* $\Gamma \vdash t.\ell_j : T$*, entonces* $\Gamma \vdash t : [\ell_i : {T_i}^{i \in 1..n}]$ *y* $T = T_j$ *para alguna* $j \in 1..n$*.*

La demostración del Lema de Sustitución para la extensión con registros se realiza para los siguientes dos casos: T_{rec} y T_{proj}.

Lema 4.3.2 (Sustitución)**.** *Si* $\Gamma, x : T_1 \vdash t : T$ *y* $\Gamma \vdash v : T_1$ *entonces* $\Gamma \vdash t[v/x] : T$*.*

Demostración. La demostración se realiza por inducción sobre la derivación de $\Gamma, x : T_1 \vdash t : T$.

Caso (T_{rec}) Considérese $\Gamma \vdash v : T_1$ y $\Gamma, x : T_1 \vdash t : T$, lo cual se obtiene de la regla (T_{rec}).
Por lo tanto, $t = [\ell_i = {s_i}^{i \in 1..n}]$, y $T = [\ell_i : {S_i}^{i \in 1..n}]$.
Entonces, la prueba es: $\Gamma \vdash [\ell_i = {s_i}^{i \in 1..n}][v/x] : [\ell_i : {S_i}^{i \in 1..n}]$.
Por inversión de la regla de tipificado, cada derivación de tipos es $\Gamma \vdash s_i : S_i$ (para $i \in 1..n$)
con la Hipótesis de Inducción: $\Gamma \vdash s[v/x] : S_i$. Aplicando la misma regla de tipificado para
$\Gamma \vdash [\ell_i = s[v/x]_i^{\,i \in 1..n}] : [\ell_i : {S_i}^{i \in 1..n}]$,
y finalmente, la sustitución de términos permite concluir el resultado.

[8] Traducido del inglés *frame typing.*
[9] Traducido del inglés *is well defined.*

Caso (T_{proj}) Supóngase $\Gamma \vdash v : T$ y $\Gamma, x : T \vdash s.\ell_j : T_j$ por demostrar que $\Gamma \vdash (s.\ell_j)\,[v/x] : T_j$.
De la regla de inversión de (T_{proj}),
la derivación $\Gamma, x : T_i \vdash s : [\ell_i : T_i \,^{i\in 1..n}]$
se tiene la siguiente Hipótesis de Inducción:
$\Gamma \vdash [\ell_i = s[v/x]_i \,^{i\in 1..n}] : [\ell_i : T_i \,^{i\in 1..n}]$, la cual por sustitución obtenemos
$\Gamma \vdash [\ell_i = s_i \,^{i\in 1..n}][v/x] : [\ell_i : T_i \,^{i\in 1..n}]$.
El resultado se mantiene al aplicar la regla (T_{proj}) a la última derivación.

■

Para demostrar que (→ **está bien definido**) [10] es necesario primero probar la siguiente proposición para registros (i.e., para los casos T_{rec} y T_{proj}):

Proposición 4.3.1. *(→ est bien definida) Si* $\Gamma \vdash t : T$, $t \to t'$ *entonces* $\Gamma \vdash t' : T'$ *donde* $T' = T$

Demostración. La prueba se realiza por inducción sobre la derivación de $\Gamma \vdash t : T$.

Caso (T_{rec}) Suponiendo que $\Gamma \vdash t : T$, donde $t = [\ell_i = t_i \,^{i\in 1..n}]$. Sin embargo, no tenemos la noción de que t' exista.

Caso (T_{proj}) Suponiendo que $\Gamma \vdash t.\ell_j : T_j$ y que existe t' tal que $t.\ell_j \to t'$
Esto significa que $t : [\ell_i = v_i \,^{i\in 1..n}]$
A su vez, la regla de reducción para registros da la proyección correspondiente $t' = v_j$ de ℓ_j.
Por Inversión, $\Gamma \vdash t : [\ell_i : T_i \,^{i\in 1..n}]$ se mantiene, por lo que por Inversión $\Gamma \vdash v_j : T_j$, lo cual también se mantiene.

■

La gramática en la Figura 4.4 presenta la semántica dinámica de STFL, usando marcos (del inglés *frames*) [95], y se extiende para registros.

$$
\begin{array}{rcll}
v & ::= & n \mid true \mid false \mid \lambda x : T.\, t \mid v :: T \mid [\ell_i = v_i \,^{i\in 1..n}] & (valores) \\
f & ::= & \Box + t \mid v + \Box \mid \mathit{if}\ \Box\ \mathit{then}\ t\ \mathit{else}\ t \mid \Box :: T \mid \Box t \mid v\Box \mid & (marcos) \\
 & & [\ell_i = v_i \,^{i\in 1..j+1}, \ell_j = \Box, \ell_k = t_k \,^{k\in j+1..n}] \mid \Box.\ell_j &
\end{array}
$$

Noción de Reducción: $\boxed{t \longrightarrow t}$

$$
\begin{array}{rcll}
n_1 + n_2 & \longrightarrow & n_3 & \text{donde } n_3 = n_1 [\![+]\!] n_2 \\
\mathit{if}\ true\ \mathit{then}\ t_1\ \mathit{else}\ t_2 & \longrightarrow & t_1 & \\
\mathit{if}\ false\ \mathit{then}\ t_1\ \mathit{else}\ t_2 & \longrightarrow & t_2 & \\
v :: T & \longrightarrow & v & \\
(\lambda x : T.\, t)\, v & \longrightarrow & t[v/x] & \\
[\ell_i = v_i \,^{i\in 1..n}].\ell_j & \longrightarrow & v_j &
\end{array}
$$

Reglas de Reducción: $\boxed{t \mapsto t}$

$$
\frac{t_1 \longrightarrow t_2}{t_1 \mapsto t_2} \qquad\qquad \frac{t_1 \mapsto t_2}{f[t_1] \mapsto f[t_2]}
$$

Figura 4.4: Semántica dinámica extendida para registros (STFL_{rec}).

[10] Del inglés (→ *is well defined*).

Los *frames*, en este caso para registros en la Figura 4.4, son utilizados en tiempo de ejecución. Para la extensión con registros, se considera que alguna etiqueta es asignada con un valor en la primer posición, mientras que el resto de las etiquetas esperan a ser inicializadas.

Para hacer una evaluación completa de un término se recurre a la definición de la cerradura transitiva (ver lado derecho de la Figura 4.5) y reflexiva (ver lado izquierdo de la Figura 4.5) de las Reglas de Reducción [66]:

Definición de Cerradura Reflexiva y Transitiva:

$$\frac{}{t \mapsto^{\star} t} \qquad\qquad \frac{t \mapsto t^{'} \qquad t^{'} \mapsto^{\star} r}{t \mapsto^{\star} r}$$

Figura 4.5: Definición de la cerradura reflexiva y transitiva [66].

Considérese las siguientes reglas de evaluación para registros:

E_{rec}**:** define la regla de evaluación para registros, donde un término t_j es reducido a un término $t_j^{'}$ en solamente un paso a través de su etiqueta en la posición j-ésima.

E_{proj}**:** define la operación de proyección, la cual se utiliza para extraer un campo de un registro, uno por uno, es decir, uno en cada tiempo. Dicho en otras palabras, dado un término t a través de su etiqueta ℓ_j se puede reducir a un término $t^{'}$ usando la misma etiqueta ℓ_j.

$E_{proj-rec}$**:** establece que si el valor de un registro es definido como $[\ell_i = {v_i}^{\,i\epsilon 1..n}]$ y ℓ_j denota a la etiqueta j-ésima, entonces el término $[\ell_i = {v_i}^{\,i\epsilon 1..n}].\ell_j$ es reducido a un valor v_j en un paso.

Ahora, se introduce la proposición de las Formas Canónicas dada por [95] extendida para registros. Esta proposición permite demostrar que STFL es seguro al agregar registros.

Proposición 4.3.2 (Formas Canónicas)**.** *Considérese un valor v tal que $\cdot \vdash v : T$. Entonces:*

1. *Si $T = Bool$ entonces $v = b$ para algún b.*
2. *Si $T = Int$ entonces $v = n$ para algún n.*
3. *Si $T = T_1 \to T_2$ entonces $v = (\lambda x : T_1.t_2)$ para algún t_2.*
4. *Si $T = [\ell_j : {T_i}^{\,i\epsilon 1..n}]$ entonces $v = [\ell_i = {v_i}^{\,i\epsilon 1..n}]$ para valores v_i.*

Ahora, considere el siguiente lema para la preservación de tipos, propuesto por [95] el cual es necesario demostrar para registros (en particular para los casos $[\ell_i = {v_i}^{\,i\epsilon 1..j+1}, \ell_j = \Box, \ell_k = {t_k}^{\,k\epsilon j+1..n}]$ y $\Box.\ell$):

Lema 4.3.3. *Considérese un frame f y el término t_1, tal que $\Gamma \vdash t_1 : T_1$ y $\Gamma \vdash f[t_1] : T$. Considérese un término t_2 tal que $\Gamma \vdash t_2 : T_2$ y $T_2 = T_1$ entonces $\Gamma \vdash f[t_2] : T^{'}$ con $T^{'} = T$.*

Demostración. La demostración se realiza por inducción sobre la derivación de $f[t_1]$.

Caso de reducción para registros. Considérese un *frame* de la forma
$f = [\ell_i = {v_i}^{\,i\epsilon 1..j+1}, \ell_j = \Box, \ell_k = {t_k}^{\,k\epsilon j+1..n}]$
y un término t' tal que $\Gamma \vdash t' : T'$ y $\Gamma \vdash f[t'] : T$,
donde $f[t'] = [\ell_i = {v_i}^{\,i\epsilon 1..j+1}, \ell_j = t', \ell_k = {t_k}^{\,k\epsilon j+1..n}]$
y $T = [\ell_i : {T_i}^{\,i\epsilon 1..n}]$.
Por la regla (T_{rec}), cada término en los campos del registro tiene un tipo, en particular el tipo de t' es T'. Ahora, considérese t'' tal que $\Gamma \vdash t'' : T''$ y $T'' = T'$. Para verificar que $\Gamma \vdash f[t''] : T$, $f[t''] = [\ell_i = {v_i}^{\,i\epsilon 1..j+1}, \ell_j = t'', l_k = {t_k}^{\,k\epsilon j+1..n}]$, donde cada tipo para los términos ${v_i}^{\,i\epsilon 1..j+1}$ y ${t_k}^{\,k\epsilon j+1..n}$ son los mismos que antes, así como el tipo del término del campo j-ésimo.

Caso de reducción para la proyección. Considérese un *frame* de la forma $f = \square.\ell$ y un término t_1 tal que $f[t_1] = t_1.\ell$, donde $T_1 = [\ell_i : T_i{}^{i\epsilon 1..n}]$ es del tipo correspondiente de t_1. Dado que ℓ es exactamente la j-ésima. Entonces por la regla de (T_{proj}), $T = T_j$. Cuando se reemplaza t_1 por t_1' en el mismo *frame*, $f[t_1^{'}] = t_1^{'}.\ell$ está también tipificado por T_{proj}. Y por hipótesis $\Gamma \vdash t_1^{'} : T_1^{'}$ y $T_1^{'} = [\ell_i : T_i^{'}{}^{i\epsilon 1..n}]$, por lo tanto $T_j' = T$.

■

A continuación se expone la Proposición 4.3.3 ($\rightarrow$ está bien definida [11]) y se realiza la demostración para los casos T_{rec} y T_{proj}.

Proposición 4.3.3 ($\rightarrow$ está bien definida)**.** *Si* $\Gamma \vdash t : T$ *and* $t \rightarrow t^{'}$ *entonces* $\Gamma \vdash t^{'} : T^{'}$*, donde* $T^{'} = T$*.*

Demostración. La demostración se realiza por inducción sobre la derivación de $\Gamma \vdash t : T$.

Caso (T_{rec}) Supóngase: $\Gamma \vdash t : T$, donde $t = [\ell_i = t_i{}^{i\epsilon 1..n}]$ sin embargo no aplica la noción de que t' exista.

Caso (T_{proj}) Supóngase: $\Gamma \vdash t.\ell_j : proj(T, \ell_j)$ (con $proj(T, \ell_j)$ definida), existe un t' tal que $t.\ell_j \mapsto t'$.
Esto significa que $t = [\ell_i = v_i{}^{i\epsilon 1..n}]$, y la regla de reducción para registro da la proyección correspondiente $t' = v_j$ del campo etiquetado para ℓ_j.
Por el Lema de Inversión 4.3.1, $\Gamma \vdash t : [\ell_i : T_i{}^{i\epsilon 1..n}]$ se mantiene, lo cual, de nuevo permite demostrar por inversión que $\Gamma \vdash v_j : T_j$.

■

A continuación se expone la Proposición 4.3.4 ($\mapsto$ *is well defined*) la cual es necesaria para la extensión con registros. Sin embargo, no es necesario demostrar la Proposición 4.3.4 para la extensión con registros, ya que la estructura original de la evaluación con *frames* no sufre ningún cambio.

Proposición 4.3.4 ($\mapsto$ está bien definida)**.** *Si* $\Gamma \vdash t : T$ *and* $t \mapsto t'$ *entonces* $\Gamma \vdash t^{'} : T^{'}$*, donde* $T^{'} = T$*.*

Demostración. Por inducción sobre la estructura de la reducción. ■

La Proposición de Seguridad [12] 4.3.5 está relacionada con la seguridad de cada término en el STFL_{rec}, que es de hecho la propiedad de progreso y preservación. Así, cualquier término cerrado es un valor o puede reducirse a otro término del mismo tipo. Por lo que es necesario probarla para cada uno de los casos de registros.

Proposición 4.3.5 (Seguridad)**.** *Si* $\cdot \vdash t : T$*, entonces uno de los siguientes casos es verdad:*

1. t *es un valor* v*.*
2. $t \mapsto t'$ $y \cdot \vdash t' : T'$ *donde* $T' = T$*.*

Demostración. La demostración se realiza por inducción sobre la estructura de t.

Caso (T_{rec}) La regla tiene el tipo $[\ell_i : T_i{}^{i\epsilon 1..n}]$ para términos $t = [\ell_i = t_i{}^{i\epsilon 1..n}]$.
Por Hipótesis de Inducción, una de las siguientes se mantiene para cada t_i:

1. t_i es un valor v_i
2. $t_i \mapsto t_i'$ con $\cdot \vdash t_i^{'} : T_i^{'}$ y $T_i^{'} = T_i$.

Entonces existen dos posibles sub-casos:

I. Cada término en un registro es un valor, entonces $t = [\ell_i = v_i{}^{i\epsilon 1..n}]$ es un valor.

[11] Del inglés *is well defined*.
[12] Se traduce en este trabajo del término en inglés *safety*.

II. Algunos términos pueden ser reducidos, entonces, sin pérdida de generalidad consideremos:
$t = [\ell_i = v_i\ ^{i\in 1..j-1}, \ell_j = t_j, \ell_k = t_k\ ^{k\in j+1..n}]$. Este término es, de hecho, un *frame*
$f = [\ell_i = v_i\ ^{i\in 1..j-}, \ell_j = \Box, \ell_k = t_k\ ^{k\in j+1..n}]$.
La Hipótesis de Inducción se aplica para un término t_j con $t_j \mapsto t'_j$.
Usando el Lema 4.3.3, se puede concluir que $\cdot \vdash [\ell_i = v_i\ ^{i\in 1..j-1}, \ell_j = t'_j, \ell_k = t_k\ ^{k\in j+1..n}] : T_j$,
y por lo tanto, existe un $t' = [\ell_i = v_i\ ^{i\in 1..j-1}, \ell_j = t'_j, \ell_k = t_k\ ^{k\in j+1..n}]$ tal que
$\cdot \vdash [\ell_i = v_i\ ^{i\in 1..j-1}, \ell_j = t'_i, \ell_k = t_k\ ^{k\in j+1..n}] : T'$ y $T' = [\ell_i : T'_i\ ^{i\in 1..n}]$.

Caso (T_{proj}) Un término $t = t_1.\ell_j$ está tipificado por la regla (T_{proj}), que es $\cdot \vdash t_1.\ell_j : proj(T, \ell_j)$ (con $proj(T, \ell_j)$ definida). Y por Lema de Inversion 4.3.1 $\cdot \vdash t_1 : [\ell_i : T_i\ ^{i\in 1..n}]$.
Por Hipótesis de Inducción, los siguientes puntos se mantienen para un término t_1:

1. t_1 es un valor. Entonces la noción de reducción para una proyección da un valor v_j el cual tiene el tipo asociado T_j.

2. Existe un término t'_1 tal que $t_1 \mapsto t'_1$, esto es, t_1 es un *frame* f y un término t_k para algún campo k : $t_1 = [\ell_i = v_i\ ^{i\in 1..k-1}, \ell_k = t_k, \ell_l = t_l\ ^{l\in k+1..n}]$.
Dado que t_1 no es un valor, entonces existe un t'_k tal que $t_k \mapsto t'_k$ y $t'_1 = [\ell_i = v_i\ ^{i\in 1..k-1}, \ell_k = t'_k, \ell_l = t_l\ ^{l\in k+1..n}]$. Usando el Lema 4.3.3 con los supuestos anteriormente descritos, la siguiente derivación de tipos se mantiene: $\Gamma \vdash f[t'_k] : [\ell_i : T_i\ ^{i\in 1..n}]$.
Finalmente, usando la regla (T_{proj}), la proyección en el campo j tiene asociado el tipo T_j.

■

En la Teoría de Tipos, la consistencia (traducido del inglés *soundness*) de un lenguaje está dada en general por dos propiedades: **progreso** y **preservación**. En algunos lenguajes, la seguridad de tipos está relacionada con programas bien tipificados que no se pueden seguir evaluando, es decir, se bloquean [13].

La propiedad de progreso dice que si un término pasa por el verificador de tipos, será para poder dar un paso en su evaluación (a menos que ya sea un valor); y la propiedad de preservación dice que el resultado de este paso tendrá el mismo tipo que el término original. Si intercalamos estos pasos (primero progreso y luego preservación tantas veces como sea necesario), podemos concluir que el resultado final, tendrá el mismo tipo que el original, por lo que se dice que el sistema de tipos es realmente consistente [57].

Un ejemplo de esto es la siguiente expresión usando notación prefija $(+\ 2\ (+\ 3\ 4))$ el cual regresará algo de tipo entero Int. En un sistema de tipos consistente, la propiedad de progreso nos dice que dados los tipos de los términos (en cada sub-expresión), éste puede dar un paso en la ejecución. Después de ese paso el programa se reduce a $(+\ 2\ \ 7)$. La propiedad de preeservación nos dirá que el tipo de la expresión es el mismo que el de la expresión original, un entero (Int), regresando el valor de 9. Como vemos la propiedad de preservación tiene el mismo tipo que la expresión previa (intermedia) Int y la propiedad de progreso se detiene pues ya tenemos una respuesta final [57].

Lo descrito en el párrafo expone que tanto la propiedad de progreso como la de preservación implican la seguridad de tipos. Sin embargo, las propiedades de progreso y preservación son mucho más fuertes que la seguridad ya que permite razonar sobre cada paso (uno por uno). En la mayoría de los lenguajes, la preservación de tipos coincide con la consistencia de tipos. Cabe mencionar, en otras disciplinas de tipos, como el Sistema F o en un lenguaje con seguridad de tipos, la consistencia de tipos corresponde a la parametricidad y a la no-interferencia, respectivamente. Además, la consistencia de tipos para un lenguaje puede ser la terminación bien definida asociada a los tipos. Depende del estudio que se esté haciendo es que se puede elegir la propiedad de interés a preservar. En esta tesis se presentan estas propiedades en la Proposición 4.3.5 y en la Proposición 4.3.3 respectivamente.

[13] Traducido del inglés *get stuck*.

Tipificado Gradual

En la metodología AGT descrita en la Subsección 4.3.1, el primer paso para realizar una extensión gradual es añadir el tipo desconocido ? [95]. La extensión gradual es una interpretación de tipos graduales.

En esta sección, se desarrolla un Lenguaje con Tipificación Gradual (GTFL) ahora extendido con registros, llamado GTFL_{rec} el cual extiende de STFL_{rec} para incluir registros así como el tipo desconocido.

$$G \in GTYPE$$

$$G ::= Int \mid Bool \mid G \to G \mid [\ell_i : G_i{}^{i\in 1..n}] \mid ?$$

La correspondencia entre STFL_{rec} y GTFL_{rec} para obtener una conexión de Galois está dada por dos funciones: $\gamma_?$ y $\alpha_?$.

La función de concretización está basada en el trabajo de Toro y Tanter [95]. La extensión con registros se define de la siguiente manera:

Definición 4.4.1 (Concretización $GTYPE$)**.** *La función de concretización $\gamma_? : GTYPE \to P(TYPE)$ está definida como sigue:*

$$\gamma_?(Int) = \{Int\}$$

$$\gamma_?(Bool) = \{Bool\}$$

$$\gamma_?(?) = TYPE$$

$$\gamma_?(G_1 \to G_2) = \{T_1 \to T_2 \mid T_1 \in \gamma_?(G_1) \wedge T_2 \in \gamma_?(G_2)\}$$

$$\gamma_?([\ell_i : G_i{}^{i\in 1..n}]) = \{[\ell_i : T_i{}^{i\in 1..n}] \mid T_i \in \gamma_?(G_i) \ \ para \ \ i \in 1..n\}$$

Observemos que la función de concretización ayuda a preservar (es decir, a conocer) la información estática de los tipos graduales [42, 95]. El caso para $\gamma_?(?) = TYPE$ establece que el tipo desconocido puede ser cualquiera de los tipos de la categoría $TYPE$ definidos en $STFL$.

La función de abstracción provee una definición para el levantamiento de conjuntos de tipos, la cual se define como $\alpha_? : P(TYPE) \rightharpoonup GTYPE$ donde el tipo de $\widehat{T}$ denota un conjunto de tipos inducidos por T en [95]:

$$\widehat{T_1 \to T_2} = \{T' \to T'' \mid T' \in \widehat{T_1} \ \wedge \ T'' \in \widehat{T_2}\}$$

$$[\ell_i : \widehat{T_i}^{\ i\in 1..n}] = \{[\ell_i : T_i'{}^{\ i\in 1..n}] \mid T_i' \in \widehat{T_i} \ \ para\, i \in 1..n\}$$

El levantamiento de tipos para registros da un conjunto de registros del mismo tamaño (n) y con el mismo orden de sus etiquetas. Los n campos de cada registro del conjunto generado tienen un tipo asociado, que es seleccionado a partir del levantamiento del correspondiente n-ésimo conjunto $\widehat{T_i}$.

La función de abstracción ayuda al lenguaje a obtener (retener) más precisión, mientras se van reduciendo la frecuencia de aparición del tipo desconocido, es decir, la función α puede producir un tipo gradual que abstrae un conjunto dado [42]. Toro y Tanter establecen que la función α es óptima, es decir, es la función con mejor aproximación, lo cual significa que se obtiene el tipo gradual más preciso que sea posible. Ambas funciones $\gamma_?$ y $\alpha_?$ se definen y aplican usando conexiones de Galois [95].

La definición para el levantamiento de un predicado es:
Sea $P \subseteq TYPE^2$ algún predicado binario sobre tipos, entonces su levantamiento (consistente) de $\widehat{P} \subseteq P(TYPE)^2$ se expresa como $\widehat{P}(\widehat{T_1}, \widehat{T_2})$ para algunos $\langle T_1, T_2\rangle \in \widehat{T_1} \times \widehat{T_2}$.

Definición 4.4.2 (Abstracción $GTYPE$). *La función de abstracción se define como:* $\alpha_? : P(TYPE) \rightharpoonup GTYPE$*:*

$$\begin{aligned}
\alpha_?\big(\{T\}\big) &= T \\
\alpha_?(\varnothing) &= \textit{indefinido} \\
\alpha_?\big(\widehat{T_1 \to T_2}\big) &= \alpha_?\big(\widehat{T}_1\big) \mathbin{\hat{\to}} \alpha_?\big(\widehat{T}_2\big) \\
\alpha_?\Big(\big[\ell_i : \widehat{T_i}^{\,i\in 1..n}\big]\Big) &= [\ell_i : \alpha_?(\widehat{T}_i)^{i\in 1..n}] \\
\alpha_?\big(\widehat{T}\big) &= ?\ \textit{en otro caso}
\end{aligned}$$

La siguiente proposición define cuando $\alpha_?$ es consistente y óptima [95]:

Definición 4.4.3 ($GTYPE$ Precisión). *G_1 es menos preciso que G_2, $G_1 \sqsubseteq G_2$, si y solo si $\gamma_?(G_1) \subseteq \gamma_?(G_2)$.*

Proposición 4.4.1 ($\alpha_?$ es consistente y óptima). *Si $\widehat{T}$ es no vacía, entonces:*

1. $\widehat{T} \sqsubseteq \gamma_?\big(\alpha_?\big(\widehat{T}\big)\big)$
2. *Si* $\widehat{T} \sqsubseteq \gamma_?(G)$ *entonces* $\alpha_?\big(\widehat{T}\big) \sqsubseteq G$.

Demostración. Para demostrar la Proposición 4.4.1 para registros, se propone una prueba formal por inducción sobre la estructura de G:

1. Considérese $\widehat{T} = [\ell_i : \widehat{T_i}^{\,i\in 1..n}] = \big\{[\ell_i : T_i{}^{\,i\in 1..n}] \mid T_i \in \widehat{T}_i \textit{ para } i \in 1..n\big\}$. Entonces, por definición de 4.4.2: $\alpha_?\Big(\big[\ell_i : \widehat{T_i}^{\,i\in 1..n}\big]\Big) = [\ell_i : \alpha_?(\widehat{T}_i)^{\,i\in 1..n}]$, y aplicando la definición para registros de 4.4.1 se tiene: $\gamma_?\big(\alpha_?\big(\widehat{T}\big)\big) = \gamma_?\big([\ell_i : \alpha_?(\widehat{T}_i)^{\,i\in 1..n}]\big) = \gamma_?\big([\ell_i : G_i{}^{\,i\in 1..n}]\big)$ con $\widehat{T}_i = G_i$ y $\gamma_?\big([\ell_i : G_i{}^{\,i\in 1..n}]\big) = \big\{[\ell_i : T_i{}^{i\in 1..n}] \mid T_i \in \gamma_?(G_i)^{\,i\in 1..n}\big\} = \big\{[\ell_i : T_i{}^{i\in 1..n}] \mid T_i \in \gamma_?(\alpha_?(\widehat{T}_i))^{\,i\in 1..n}\big\}$.
   Por Hipótesis de Inducción: $T_i \in \widehat{T}_i \sqsubseteq \gamma_?\big(\alpha_?\big(\widehat{T}_i\big)\big)$. Por lo tanto, para cada T_{ij}, el resultado se mantiene.

2. Sea $\widehat{T} = [\ell_i : \widehat{T_i}^{\,i\in 1..n}]$, y supóngase $\widehat{T} = [\ell_i : \widehat{T_i}^{\,i\in 1..n}] \sqsubseteq \gamma_?\big([\ell_i : G_i{}^{\,i\in 1..n}]\big)$.
   Por definición de $\gamma_?$: $\big\{[\ell_i : T_i{}^{\,i\in 1..n}] \mid T_i \in \gamma_?(G_i)^{\,i\in 1..n}\big\}$, y por definición de $\alpha_?$ para registros: $\alpha_?\Big(\big[\ell_i : \widehat{T_i}^{\,i\in 1..n}\big]\Big) = [\ell_i : \alpha_?\big(\widehat{T}_i\big)^{\,i\in 1..n}]$.
   Por Hipótesis de Inducción, para cada $\widehat{T}_i$ si $\widehat{T}_i \sqsubseteq \gamma_?(G_i)$ entonces $\alpha_?\big(\widehat{T}_i\big) \sqsubseteq G_i$.
   Por lo tanto, usando definición de 4.4.3: $\gamma_?\Big(\alpha_?\Big(\big[\ell_i : \widehat{T_i}^{\,i\in 1..n}\big]\Big)\Big) = \gamma_?\big([\ell_i : \alpha_?\big(\widehat{T}_i\big)^{\,i\in 1..n}]\big) \sqsubseteq \gamma_?\big([\ell_i : G_i{}^{\,i\in 1..n}]\big)$, el resultado se mantiene.

■

El siguiente paso para dar la interpretación estratificada del tipificado de unión gradual es hacer un levantamiento de funciones, las cuales son $\alpha_?$ y $\gamma_?$ a conjuntos finitos, esto es de $P_{fin}(P(TYPE))$ a $P_{fin}(GTYPE)$ y viceversa. Es importante mencionar que este levantamiento se hace sobre conjuntos finitos de tipos, por medio de otra conexión de Galois, donde $\widehat{G}$ denota a un conjunto de tipos graduales [95].

Definición 4.4.4 (Concretización $P_{fin}(GTYPE)$).

$$\widehat{\gamma_?} : P_{fin}(GTYPE) \to P_{fin}(P(TYPE))$$

está definida como:

$$\widehat{\gamma_?}\big(\widehat{G}\big) = \big\{\gamma_?(G) \mid G \in \widehat{G}\big\}$$

Definición 4.4.5 (Abstracción $P_{fin}\left(GTYPE\right)$). $\widehat{\alpha_?}: P_{fin}\left(P\left(TYPE\right)\right) \rightharpoonup P_{fin}\left(GTYPE\right)$ *está definida como:*

$$\widehat{\alpha_?}(\varnothing) = \textit{indefinida} \qquad \widehat{\alpha_?}\left(\widehat{\widehat{T}}\right) = \bigcup_{\widehat{T} \in \widehat{\widehat{T}}} \alpha_?\left(\widehat{T}\right)$$

De nuevo, este par de funciones describen la mejor aproximación.

Proposición 4.4.2 ($\widehat{\alpha_?}$ es consistente y óptima). *Si* $\widehat{\widehat{T}}$ *es no vacía, entonces:*

1. $\widehat{\widehat{T}} \sqsubseteq \widehat{\gamma_?}\left(\widehat{\alpha_?}\left(\widehat{\widehat{T}}\right)\right)$
2. $\widehat{\widehat{T}} \sqsubseteq \widehat{\gamma_?}\left(\widehat{G}\right)$ *entonces* $\widehat{\alpha_?}\left(\widehat{\widehat{T}}\right) \sqsubseteq \widehat{G}$

Demostración. La demostración es la misma que la expuesta por Toro y Tanter [95] dado que estamos trabajando también sobre conjuntos. ■

Una observación importante es que $\widehat{\widehat{T}}$ es, de hecho, del tipo de $\widehat{G}$.

Tipificado de Unión Gradual

El lenguaje gradual $\mathsf{GTFL}^{\oplus}_{rec}$ incluye las Uniones Graduales como un tipo de constructor de tipos para extender el sistema de tipos de GTFL_{rec}. Las conexiones de Galois entre las categorías sintácticas de tipos se presentan dado el enfoque estratificado. Primero, los tipos incluyen las Uniones Graduales, y posteriormente, se realiza la combinación de los dos enfoques: el tipo desconocido y las Uniones Graduales. Esta sección describe el desarrollo de $\mathsf{GTFL}^{\oplus}_{rec}$, su definición, semántica y las demostraciones de sus propiedades, en particular la Garantía Gradual Dinámica. El *Thresome Calculus* se utiliza entonces como la última traducción para lograr una semántica traslacional (del inglés *translational semantics*).

Unión e Interpretación Estratificada

En esta sección se presenta una conexión de Galois entre $GTYPE$, $P\left(TYPE\right)$ y $P_{fin}\left(GTYPE\right)$. Aquí se presentan las uniones graduales para un lenguaje con registros. El lenguaje $\mathsf{GTFL}^{\oplus}_{rec}$ distingue dos colecciones de tipos. Por un lado, $STYPE$ maneja tipos graduales utilizando únicamente uniones graduales. Y por otro lado, $UTYPE$ combina ambos constructores de tipos graduales: el tipo desconocido ($?$) y la unión ($\oplus$).

La última conexión de Galois para el enfoque estratificado de AGT define una unión que es la combinación de Tipos Graduales y Tipos Unión, es decir, es el conjunto de la unión de interpretaciones [95]. La AGT permite construir la semántica de los Tipos Graduales a partir de conjuntos preexistentes de tipos estáticos. Esta conexión, dada por $\gamma_{\oplus}$ y $\alpha_{\oplus}$, la cual relaciona los tipos de Unión Gradual $UTYPE$ con conjuntos finitos de conjuntos de tipos estáticos $P_{fin}\left(GTYPE\right)$.

Estas funciones se dan para ambas colecciones de tipos, $STYPE$ y $UTYPE$.

Para construir el Tipificado de Unión Gradual, se propone la gramática STYPE dada por [95]. Esta gramática se compone solo de uniones graduales sin el tipo desconocido. La gramática $STYPE$ extiende los tipos en STFL_{rec} con uniones graduales.

$$S \in STYPE$$

$$S ::= Int \mid Bool \mid S \to S \mid [\ell_i : S_i{}^{i \in 1..n}] \mid S \oplus S$$

Los tipos estáticos como STYPE contienen tipos de enteros (Int), booleanos ($Bool$), funciones ($S \to S$), las uniones graduales denotadas por $S \oplus S$ y el tipo registro (*record*).

Las Uniones Graduales representan un conjunto finito de tipos. Su interpretación se obtiene a través de la función de concretización definida sólo para producir conjuntos finitos de tipos estáticos. La función de abstracción produce una unión gradual de tipos estáticos denotada como $\oplus$ [95].

Definición 4.5.1 (Concretización $STYPE$). $\gamma_\oplus : STYPE \to P_{fin}(TYPE)$

$$\gamma_\oplus(Int) = \{Int\}$$

$$\gamma_\oplus(Bool) = \{Bool\}$$

$$\gamma_\oplus(S_1 \to S_2) = \{T_1 \to T_2 \mid T_1 \in \gamma_\oplus(S_1) \ \wedge \ T_2 \in \gamma_\oplus(S_2)\}$$

$$\gamma_\oplus(S_1 \oplus S_2) = \gamma_\oplus(S_1) \ \cup \ \gamma_\oplus(S_2)$$

$$\gamma_\oplus([\ell_i : S_i{}^{i\in 1..n}]) = \{[\ell_i : T_i{}^{i\in 1..n}] \mid T_i \in \gamma_\oplus(S_i) \ \ para\ i \in 1..n\}$$

Definición 4.5.2 (Abstracción $STYPE$). *La función de abstracción $\alpha_\oplus : P_{fin}(TYPE) \rightharpoonup STYPE$ se define como sigue:*

$$\alpha_\oplus(T) = \bigoplus T \qquad si\ T \neq \varnothing$$

donde $\bigoplus T$ es la Unión Gradual de todos los tipos en el conjunto finito T.

Las uniones graduales representan un conjunto finito de tipos [95], donde las funciones tanto de concretización como de abstracción están definidas entre tipos estáticos y conjuntos finitos de tipos generados por las conexiones de Galois.

Para la interpretación gradual de uniones de tipos con registros, el siguiente paso es diseñar los tipos de unión gradual usando el AGT para construir la semántica para tipos graduales en términos de conjuntos de tipos estáticos pre-existentes.

El nuevo sistema de tipos está compuesto de Uniones Graduales [14] $\oplus$, llamado GTFL$_\oplus$. En éste se combinan ambos constructores, STYPE y UTYPE. La última categoría sintáctica, UTYPE, es una extensión que incluye el tipo desconocido, las uniones graduales y los registros.

La gramática correspondiente para tipos de $\mathsf{GTFL}^{\oplus}_{rec}$ es:

$$U \in UTYPE$$

$$U ::= Bool \mid Int \mid U \to U \mid [\ell_i : U_i{}^{i\in 1..n}] \mid ? \mid U \oplus U$$

Definición 4.5.3 (Concretización $UTYPE$). *La concretización $\gamma_\oplus : UTYPE \to P_{fin}(GTYPE)$ es definida como:*

$$\gamma_\oplus(Int) = \{Int\}$$

$$\gamma_\oplus(Bool) = \{Bool\}$$

$$\gamma_\oplus(?) = \{?\}$$

$$\gamma_\oplus(U_1 \to U_2) = \{T_1 \to T_2 \mid T_1 \in \gamma_\oplus(U_1) \wedge T_2 \in \gamma_\oplus(U_2)\}$$

$$\gamma_\oplus(U_1 \oplus U_2) = \gamma_\oplus(U_1) \ \cup \ \gamma_\oplus(U_2)$$

$$\gamma_\oplus([\ell_i : U_i{}^{i\in 1..n}]) = \{[\ell_i : T_i{}^{i\in 1..n}] \mid T_i \in \gamma_\oplus(U_i) \ \ para\ i \in 1..n)\}$$

Definición 4.5.4 (Abstracción $UTYPE$). *La abstracción $\alpha_\oplus : P_{fin}(GTYPE) \rightharpoonup UTYPE$ es definida como:*

$$\alpha_\oplus(\widehat{G}) = \bigoplus \widehat{G} \quad \ if\ \widehat{G} \neq \varnothing$$

donde $\bigoplus \widehat{G}$ es la Unión Gradual de todos los tipos en el conjunto $\widehat{G}$.

La definición de la función de abstracción $\alpha_\oplus$ (abstracción UTYPE) es la misma que define Toro y Tanter en [95] y produce conjuntos finitos. Esto es la unión gradual de todos los elementos en $\widehat{G}$.

La siguiente proposición establece que $\alpha_\oplus$ es consistente y óptima para UTYPE. La noción de precisión para UTYPE es la misma que la establecida anteriormente, esto es que $U_1 \sqsubseteq U_2$ si y sólo si $\gamma_\oplus(U_1) \subseteq \gamma_\oplus(U_2)$

[14] Traducción de *Gradual Unions*.

Proposición 4.5.1 (Abstracción $UTYPE$ es consistente y óptima)**.** *Si $\widehat{G}$ es no vacía, entonces:*

1. $\widehat{G} \sqsubseteq \gamma_{\oplus}(\alpha_{\oplus}(\widehat{G}))$
2. *Si* $\widehat{G} \sqsubseteq \gamma_{\oplus}(U)$ *entonces* $\alpha_{\oplus}(\widehat{G}) \sqsubseteq U$

Demostración. La demostración se realiza por inducción sobre la estructura de G y U respectivamente. El siguiente caso se presenta para registros:

1. Supóngase $G = [\ell_i : {G_i}^{i\in 1..n}]$ entonces $\widehat{G} = \{[\ell_i : {G'_i}^{i\in 1..n}] \mid G'_i \in \widehat{G_i}\ para\ i \in 1..n\}$ donde s es el tamaño de $\widehat{G}$.
   Para demostrar que $\widehat{G} \sqsubseteq \gamma_{\oplus}(\alpha_{\oplus}(\widehat{G}))$, se procede aplicando la definición de $\alpha_{\oplus}$:
   $\alpha_{\oplus}\Big(\{[\ell_i : {G'_i}^{i\in 1..n}] \mid G'_i \in \widehat{G_i}\ for\ i \in 1..n\}\Big) = \bigoplus\{[\ell_i : {G'_i}^{i\in 1..n}] \mid G'_i \in \widehat{G_i}\ for\ i \in 1..n\} =$
   $\{[\ell_1 : \alpha_{\oplus}(G'_1), \ldots, \ell_n : \alpha_{\oplus}(G_n)] \mid G'_1 \in \widehat{G_1}\} \oplus \ldots \oplus \{[\ell_1 : \alpha_{\oplus}(G_1), \ldots \ell_n : \alpha_{\oplus}(G'_n)] \mid G'_n \in \widehat{G_n}\}$.
   Usando la definición de $\gamma_{\oplus}$:
   $\gamma_{\oplus}\Big(\bigoplus\{[\ell_i : {G'_i}^{i\in 1..n}] \mid G'_i \in \widehat{G_i}\ for\ i \in 1..n\}\Big) = \bigcup_s \gamma_{\oplus}\Big(\bigoplus\{[\ell_i : {G'_i}^{i\in 1..n}] \mid G'_i \in \widehat{G_i}\ for\ i \in 1..n\}\Big) =$
   $\{[\ell_1 : \gamma_{\oplus}(\alpha_{\oplus}(G'_1)), \ldots, \ell_n : \gamma_{\oplus}(\alpha_{\oplus}(G_n))] \mid G'_1 \in \widehat{G_1}\} \cup \ldots \cup \{[\ell_1 : \gamma_{\oplus}(\alpha_{\oplus}(G_1)), \ldots \ell_n : \gamma_{\oplus}(\alpha_{\oplus}(G'_n))] \mid G'_n \in \widehat{G_n}\}$.
   Por Hipótesis de Inducción, cada $T_i \in \widehat{G_i}$ y además pertence a $\gamma_{\oplus}(\alpha_{\oplus}(\widehat{G_i}))$ para $i \in 1..n$ y por lo tanto los registros en $\widehat{G}$ están incluídos $\gamma_{\oplus}(\alpha_{\oplus}(\widehat{G}))$.
2. Supóngase que $\widehat{G} \sqsubseteq \gamma_{\oplus}(U)$, donde $G = [\ell_i : {G_i}^{i\in 1..n}]$ tal que
   $\widehat{G} = \{[\ell_i : {G'_i}^{i\in 1..n}] \mid G'_i \in \widehat{G_i}\ for\ i \in 1..n\} \subseteq \gamma_{\oplus}(U)$ y $\gamma_{\oplus}(U) = \gamma_{\oplus}\big([\ell_i : {U_i}^{i\in 1..n}]\big) = \{[\ell_i : {T_i}^{i\in 1..n}] \mid T_i \in \gamma_{\oplus}(U_i)\ para\ i \in 1..n\}$.
   Para demostrar que $\alpha_{\oplus}(\widehat{G}) \sqsubseteq U$, se debe proceder usando la definición de $\alpha_{\oplus}(\widehat{G})$:
   $\alpha_{\oplus}([\ell_i : \widehat{{G_i}^{i\in 1..n}}]) = \bigoplus\{[\ell_i : {G'_i}^{i\in 1..n}] \mid G'_i \in \widehat{G_i}\ for\ i \in 1..n\} =$
   $\{[\ell_1 : \alpha_{\oplus}(G'_1), \ldots, \ell_n : \alpha_{\oplus}(G_n)] \mid G'_1 \in \widehat{G_1}\} \oplus \ldots \oplus \{[\ell_1 : \alpha_{\oplus}(G_1), \ldots \ell_n : \alpha_{\oplus}(G'_n)] \mid G'_n \in \widehat{G_n}\}$.
   Entonces, por definición de $\sqsubseteq$ la demostración se dirige a
   $\gamma_{\oplus}\Big(\{[\ell_1 : \alpha_{\oplus}(G'_1), \ldots, \ell_n : \alpha_{\oplus}(G_n)] \mid G'_1 \in \widehat{G_1}\} \oplus \ldots \oplus \{[\ell_1 : \alpha_{\oplus}(G_1), \ldots \ell_n : \alpha_{\oplus}(G'_n)] \mid G'_n \in \widehat{G_n}\}\Big) =$
   $\{[\ell_1 : \gamma_{\oplus}(\alpha_{\oplus}(G'_1)), \ldots, \ell_n : \gamma_{\oplus}(\alpha_{\oplus}(G_n))] \mid G'_1 \in \widehat{G_1}\} \cup \ldots \cup$
   $\{[\ell_1 : \gamma_{\oplus}(\alpha_{\oplus}(G_1)), \ldots \ell_n : \gamma_{\oplus}(\alpha_{\oplus}(G'_n))] \mid G'_n \in \widehat{G_n}\} \subseteq \gamma_{\oplus}(U)$ lo cual mantiene la Hipótesis de Inducción para cada U_i, si $\widehat{G_i} \sqsubseteq \gamma_{\oplus}(U_i)$ entonces $\alpha_{\oplus}(\widehat{G_i}) \sqsubseteq U_i$ la cual por definición de precisión da $\gamma_{\oplus}(\alpha_{\oplus}(\widehat{G_i})) \subseteq \gamma_{\oplus}(U_i)$.

■

Por tanto, en UTYPE se define en términos de conjuntos de conjuntos de tipos estáticos con la composición de dos conexiones de Galois. Esto se expresa mediante las definiciones de las funciones de concretización y abstracción para definir la última conexión de Galois de la aproximación estratificada a AGT.

Siguiendo la metodología de Toro y Tanter [95], la interpretación estratificada de UTYPE está definida en términos de conjuntos de conjuntos de tipos estáticos con la composición de dos conexiones de Galois:

Definición 4.5.5 (Concretización)**.** $\gamma : UTYPE \to P_{fin}(P(TYPE))$*, definida como* $\gamma = \widehat{\gamma_?} \circ \gamma_{\oplus}$

Definición 4.5.6 (Abstracción)**.** $\alpha : P_{fin}(P(TYPE)) \rightharpoonup UTYPE$*, definida como* $\alpha = \alpha_{\oplus} \circ \widehat{\alpha_?}$

La siguiente proposición es necesaria para probar que α es consistente y óptima:

Proposición 4.5.2 (α es consistente y óptima)**.** *Si $\widehat{\widehat{T}}$ es no vacía, entonces:*

1. $\widehat{\widehat{T}} \sqsubseteq \gamma(\alpha(\widehat{\widehat{T}}))$
2. *Si* $\widehat{\widehat{T}} \sqsubseteq \gamma(U)$ *entonces* $\alpha(\widehat{\widehat{T}}) \sqsubseteq U$

Demostración. Esta proposición se mantiene por la composición de la abstracción consistente y óptima. Éstas han sido demostradas en las Proposiciones 4.4.2 y 4.5.1. ∎

El siguiente paso es desarrollar $\mathsf{GTFL}^{\oplus}_{rec}$, en el cual cada tipo U es levantado (del inglés *is lifted*) de los términos graduales $t \in UTERM$, con sus respectivas anotaciones de tipo gradual asociadas. Esta nueva categoría estratificada utiliza la última conexión de Galois entre conjuntos de conjuntos de tipos estáticos [95].
La Figura 4.6 muestra la gramática del sistema completo de $\mathsf{GTFL}^{\oplus}_{rec}$. Se usa $\widetilde{t}$ para denotar a los términos en este sistema.

$$U \in UTYPE, \quad x \in VAR, \quad \widetilde{t} \in UTERM,$$

$$\Gamma \in VAR \xrightarrow{\text{fin}} UTYPE, \quad \ell \in ULABEL$$

$$\begin{array}{rcll} U & ::= & Bool \mid Int \mid U \to U \mid [\ell_i : U_i{}^{i\in 1..n}] \mid ? \mid U \oplus U & \text{(tipos)} \\ v & ::= & n \mid true \mid false \mid \lambda x : U.\widetilde{t} \mid [\ell_i = v_i{}^{i\in 1..n}] & \text{(valores)} \\ \widetilde{t} & ::= & v \mid x \mid \widetilde{t} + \widetilde{t} \mid \mathit{if}\ \widetilde{t}\ \mathit{then}\ \widetilde{t}\ \mathit{else}\ \widetilde{t} \mid \widetilde{t} :: U \mid \widetilde{t}\,\widetilde{t} \mid [\ell_i = \widetilde{t}_i{}^{i\in 1..n}] \mid \widetilde{t}.\ell_j & \text{(términos)} \end{array}$$

$$\frac{}{\Gamma \vdash n : Int}\ (U_n) \qquad \frac{}{\Gamma \vdash b : Bool}\ (U_b) \qquad \frac{x : U \in \Gamma}{\Gamma \vdash x : U}\ (U_x)$$

$$\frac{\Gamma \vdash \widetilde{t_1} : U_1 \qquad U_1 \sim Int \qquad \Gamma \vdash \widetilde{t_2} : U_2 \qquad U_2 \sim Int}{\Gamma \vdash \widetilde{t_1} + \widetilde{t_2} : Int}\ (U_+)$$

$$\frac{\Gamma \vdash \widetilde{t_1} : U_1 \qquad U_1 \sim Bool \qquad \Gamma \vdash \widetilde{t_2} : U_2 \qquad \Gamma \vdash \widetilde{t_3} : U_3}{\Gamma \vdash \mathit{if}\ \widetilde{t_1}\ \mathit{then}\ \widetilde{t_2}\ \mathit{else}\ \widetilde{t_3} : U_2 \sqcap U_3}\ (U_{if}) \qquad \frac{\Gamma \vdash \widetilde{t} : U \qquad U \sim U_1}{\Gamma \vdash (\widetilde{t} :: U_1) : U_1}\ (U_{::})$$

$$\frac{\Gamma, x : U_1 \vdash \widetilde{t} : U_2}{\Gamma \vdash (\lambda x : U_1.\widetilde{t}) : U_1 \to U_2}\ (U_\lambda) \qquad \frac{\Gamma \vdash \widetilde{t_1} : U_1 \qquad \Gamma \vdash \widetilde{t_2} : U_2 \qquad U_2 \sim \widetilde{dom}(U_1)}{\Gamma \vdash \widetilde{t_1}\,\widetilde{t_2} : \widetilde{cod}(U_1)}\ (U_{app})$$

$$\frac{\Gamma \vdash \widetilde{t_i} : U_i \qquad \textit{para cada } i \in 1..n}{\Gamma \vdash [\ell_i = \widetilde{t}_i{}^{i\in 1..n}] : [\ell_i : U_i{}^{i\in 1..n}]}\ (U_{rec}) \qquad \frac{\Gamma \vdash \widetilde{t} : U \qquad U \sim [\ell_i : U_i{}^{i\in 1..n}]}{\Gamma \vdash \widetilde{t}.\ell_j : \widetilde{proj}(U, \ell_j)}\ (U_{proj})$$

$$\begin{array}{c} \widetilde{dom} : UTYPE \rightharpoonup UTYPE \\ \widetilde{cod} : UTYPE \rightharpoonup UTYPE \\ \widetilde{proj} : UTYPE \times LABEL \rightharpoonup UTYPE \end{array} \qquad \begin{array}{c} \widetilde{dom}(U) = \alpha\big(\widehat{dom}(\gamma(U))\big) \\ \widetilde{cod}(U) = \alpha\big(\widehat{cod}(\gamma(U))\big) \\ \widetilde{proj}(U, \ell_j) = \alpha\big(\widehat{proj}(\gamma(U), \ell_j)\big) \end{array}$$

Figura 4.6: Sistema del Lenguaje Funcional de Tipos Gradual, $\mathsf{GTFL}^{\oplus}_{rec}$

Como es un lenguaje gradual, la relación de consistencia debe estar definida para $UTYPE$ (denotada por $U_1 \sim U_2$) la cual se establece por la siguiente definición. Obsérvese que la consistencia de los registros se verifica puntualmente (del inglés *pointwise*).

A continuación, se exponen algunos ejemplos usando derivaciones de tipos en $\mathsf{GTFL}^{\oplus}_{rec}$ definidos en la Figura 4.6.

Primer ejemplo usando (U_+)

$$\dfrac{\dfrac{}{\Gamma \vdash 2 : Int}\,(U_n) \qquad Int \sim Int \qquad \dfrac{}{\Gamma \vdash 3 : Int}\,(U_n) \qquad Int \sim Int}{\Gamma \vdash 2 + 3 : Int}\,(U_+)$$

Segundo ejemplo usando (U_{if})

$$\dfrac{\dfrac{}{\Gamma \vdash true : Bool}\,(U_b) \qquad Bool \sim Bool \qquad \dfrac{}{\Gamma \vdash 1 : Int}\,(U_n) \qquad \dfrac{}{\Gamma \vdash 2 : Int}\,(U_n)}{\Gamma \vdash if\ true\ then\ 1\ else\ 2 : Int \sqcap Int}\,(U_{if})$$

Tercer ejemplo usando $(U_{app}$, U_λ, U_b, U_{proj}, U_{rec}, U_n y $U_b)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : Int \oplus Bool \vdash 1 : Int}\,(U_n) \quad \dfrac{}{\Gamma, x : Int \oplus Bool \vdash true : Bool}\,(U_b)}{\Gamma, x : Int \oplus Bool \vdash [\ell_1 = 1, \ell_2 = true] : [\ell_1 : Int, \ell_2 : Bool]}\,(U_{rec}) \quad [\ell_1 : Int, \ell_2 : Bool] \sim [\ell_1 : Int, \ell_2 : Bool]}{\Gamma, x : Int \oplus Bool \vdash [\ell_1 = 1, \ell_2 = true].\ell_2 : Bool}\,(U_{proj})}{\Gamma \vdash \lambda x : Int \oplus Bool.\,([\ell_1 = 1, \ell_2 = true].\ell_2) : Int \oplus Bool \to Bool}\,(U_\lambda) \quad \dfrac{}{\Gamma \vdash true : Bool}\,(U_b) \quad Bool \sim Int \oplus Bool}{\Gamma \vdash ((\lambda x : Int \oplus Bool.\,([\ell_1 = 1, \ell_2 = true].\ell_2))\ true) : Bool}\,(U_{app})$$

Cuarto ejemplo usando $(U_{app}$, U_λ, U_x, U_{proj}, U_{rec}, U_n y $U_b)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : Int \oplus ? \vdash 1 : Int}\,(U_n) \quad \dfrac{}{\Gamma, x : Int \oplus ? \vdash true : Bool}\,(U_b)}{\Gamma, x : Int \oplus ? \vdash [\ell_1 = 1, \ell_2 = true] : [\ell_1 : Int, \ell_2 : Bool]}\,(U_{rec}) \quad [\ell_1 : Int, \ell_2 : Bool] \sim [\ell_1 : Int, \ell_2 : Bool]}{\Gamma, x : Int \oplus ? \vdash [\ell_1 = 1, \ell_2 = true].\ell_2 : Bool}\,(U_{proj})}{\Gamma \vdash \lambda x : Int \oplus ?.\,([\ell_1 = 1, \ell_2 = true].\ell_2) : Int \oplus ? \to Bool}\,(U_\lambda) \quad \dfrac{y : ? \in \Gamma}{\Gamma \vdash y : ?}\,(U_x) \quad Bool \sim Int \oplus ?}{\Gamma \vdash ((\lambda x : Int \oplus ?.\,([\ell_1 = 1, \ell_2 = true].\ell_2))\ y) : Bool}\,(U_{app})$$

Figura 4.7: Ejemplos de derivación de tipos de $\mathsf{GTFL}^{\oplus}_{rec}$.

Definición 4.5.7 (Relación de Consistencia Gradual)**.** *Dos tipos graduales son consistentes, $U_1 \sim U_2$, si y sólo si existe $T_1 \in \gamma(U_1)$, $T_1' \in T_1$, y $T_2 \in \gamma(U_2)$, $T_2' \in T_2$ tal que $T_1' = T_2'$.*
Inductivamente:

$$\dfrac{}{U \sim U} \qquad \dfrac{}{? \sim U} \qquad \dfrac{}{U \sim ?}$$

$$\dfrac{U \sim U_1}{U \sim U_1 \oplus U_2} \qquad \dfrac{U \sim U_2}{U \sim U_1 \oplus U_2} \qquad \dfrac{U_1 \sim U}{U_1 \oplus U_2 \sim U} \qquad \dfrac{U_2 \sim U}{U_1 \oplus U_2 \sim U}$$

$$\dfrac{U_{21} \sim U_{11} \qquad U_{12} \sim U_{22}}{U_{11} \to U_{12} \sim U_{21} \to U_{22}} \qquad \dfrac{U_{11} \sim U_{21} \qquad \cdots \qquad U_{1n} \sim U_{2n}}{[\ell_1 : U_{11}, \ldots, \ell_n : U_{1n}] \sim [\ell_1 : U_{21}, \ldots, \ell_n : U_{2n}]}$$

Para seguir con la exposición de la semántica dinámica del Tipificado de Unión Gradual se brinda la siguiente definición para *Gradual Meet* [95], la cual incluye el caso para registros. Este caso se define únicamente para obtener el *Gradual Meet* de dos registros del mismo tamaño con las mismas etiquetas ordenadas.

Definición 4.5.8 (Gradual Meet)**.** *Sea* $\sqcap : UTYPE \times UTYPE \rightharpoonup UTYPE$ *definida como:*

1. $U \sqcap U = U$
2. $? \sqcap U = U \sqcap ? = U$
3. $U \sqcap (U_1 \oplus U_2) = (U_1 \oplus U_2) \sqcap U = \begin{cases} U \sqcap U_1 & \text{si } U \sqcap U_2 \text{ es indefinida} \\ U \sqcap U_2 & \text{si } U \sqcap U_1 \text{ es indefinida} \\ (U \sqcap U_1) \sqcap (U \sqcap U_2) & \text{en otro caso} \end{cases}$
4. $(U_{11} \to U_{12}) \sqcap (U_{21} \to U_{22}) = (U_{11} \sqcap U_{21}) \to (U_{12} \sqcap U_{22})$
5. $[\ell_1 : U_{11}, ..., \ell_n : U_{1n}] \sqcap [\ell_1 : U_{21}, ..., \ell_n : U_{2n}] = [\ell_1 : (U_{11} \sqcap U_{21}), \ldots, \ell_n : (U_{1n} \sqcap U_{2n})]$
6. $U_1 \sqcap U_2$ *en otro caso es indefinida*

La consistencia de los tipos debe ser preservada por los predicados y las funciones. Por lo tanto, se define a continuación un levantamiento de STFL_{rec} a $\mathsf{GTFL}^{\oplus}_{rec}$, siguiendo [95]. Recuérdese que en la metodología AGT, la última conexión relaciona los tipos graduales con conjuntos de tipos estáticos dados por las funciones γ y α (véanse las definiciones 4.5.5 y 4.5.6).

Definición 4.5.9 (Levantamiento de predicados y funciones)**.**

- *Sea un predicado* P*, el levantamiento gradual* $\widetilde{P} \in UTYPE^2$*,* $\widetilde{P}(U_1, U_2)$ *se mantiene, si y sólo si existe* $T_1 \in \gamma(U_1)$ *y* $T_2 \in \gamma(U_2)$ *tal que* $P(T_1, T_2)$*.*
- *Para una función dada* f*, el levantamientos usa la aplicación punto a punto (pointwise application) de* f *para todos los elementos del conjunto potencia:* $\widetilde{f} = \alpha \circ f \circ \gamma$

La siguiente definición establece que la traducción de *equate* de STFL a una función de $\text{GTFL}_{\oplus}$, para preservar el tipificado es:

Definición 4.5.10 (Levantamiento *equate*)**.**

$$\begin{aligned} \widetilde{equate}(U_1, U_2) &= U_1 \sqcap U_2 \\ &= \alpha(\{\widehat{T_1} \sqcap \widehat{T_2} \mid \widehat{T_1} \in \gamma(U_1), \widehat{T_2} \in \gamma(U_2)\}) \\ &= \alpha_{\oplus}(\{\widetilde{equate_?}(G_1, G_2) \mid G_1 \in \gamma_{\oplus}(U_1), G_2 \in \gamma_{\oplus}(U_2)\}) \end{aligned}$$

La siguiente proposición es necesaria para demostrar la proposición de *Gradual Meet* [15] para $GTFL_{\oplus}$ siguiendo [95]:

Proposición 4.5.3 (Gradual Meet)**.** *Gradual Meet se define como la composición de la función de abstracción, la función equate y la función de concretización.*

$$\sqcap = \alpha \circ \widetilde{\widetilde{equate}} \circ \gamma$$

Demostración. La demostración se realiza usando inducción estructural y la definición de *gradual meet* como intersección de conjuntos de conjuntos [95]. El caso para registros es directo debido a que *gradual meet* es punto a punto. ■

Como la interpretación gradual requiere una relación de precisión, ésta se define a continuación para los tipos de unión gradual.

Proposición 4.5.4 (Precisión de tipos)**.**

$$\frac{}{U \sqsubseteq U} \qquad \frac{}{U \sqsubseteq ?} \qquad \frac{U_1 \sqsubseteq U_3 \qquad U_2 \sqsubseteq U_4}{U_1 \to U_2 \sqsubseteq U_3 \to U_4}$$

$$\frac{U_2 \sqsubseteq U_1 \qquad U_3 \sqsubseteq U_1}{U_2 \oplus U_3 \sqsubseteq U_1} \qquad \frac{U_3 \sqsubseteq U_1}{U_3 \sqsubseteq U_1 \oplus U_2} \qquad \frac{U_3 \sqsubseteq U_2}{U_3 \sqsubseteq U_1 \oplus U_2} \qquad \frac{U_{11} \sqsubseteq U_{21} \qquad \cdots \qquad U_{1n} \sqsubseteq U_{2n}}{[\ell_1 : U_{11}, \ldots, \ell_n : U_{1n}] \sqsubseteq [\ell_1 : U_{21}, \ldots, \ell_n : U_{2n}]}$$

[15] La traducción de *Gradual Meet* es máxima cota inferior gradual.

Demostración. La demostración es directa usando inducción y la función de concretización $\gamma_{\oplus}$. ■

La semántica dinámica de $\mathsf{GTFL}^{\oplus}_{rec}$ utiliza términos intrínsecos. Este enfoque se basa en la definición de términos intrínsecos de Church [24] donde cualquier término bien definido tiene asociado un único tipo [65]. Por lo tanto, un tipo es un atributo esencial (intrínseco) de un término dado por la correspondencia directa entre las derivaciones de términos y sus tipos [42].

En la metodología AGT se establece que la semántica dinámica directa utiliza la reducción en derivaciones de tipo graduales a través de términos intrínsecos. Un término intrínseco se denota como $\breve{t}$ [95]. Para representar el juicio de tipo gradual $\Gamma \vdash \widetilde{t} : U$, aquí la notación t^U hace explícito un término intrínseco y algunas veces se prefiere denotar como $\breve{t} \in TERM_U$ por facilidad en el uso de la notación. Los términos intrínsecos son definidos en [42] como una "familia de conjuntos de tipos indexados".

Los términos intrínsecos necesitan incluir una **evidencia**, denotada como ε, la cual refina los juicios de tipos consistentes en una derivación. Una evidencia es un testigo que caracteriza los posibles tipos estáticos de un término para recuperar la información precisa en un momento dado. En cualquier otro caso se envía un error. Por lo tanto, la evidencia es información en tiempo de ejecución que se utiliza para apoyar la tipificación gradual consistente y la tipificación gradual estática, la cual es además información que cambia durante la ejecución.

Por lo tanto, cuando un término intrínseco se reduce a un valor resultante, es necesario tener suficiente evidencia, obtenida a través de los juicios de tipos consistentes para reforzar el tipo obtenido [42], ya que ha evolucionado monótonamente mediante la transitividad de la evidencia consistente a partir de evidencia inicial.

Las evidencias ayudan a soportar la relación de consistencia de tipos gradual. Además, la evidencia mejora la semántica dinámica para preservar la seguridad en cada paso de la reducción, o eventualmente, dar un error en tiempo de ejecución [11, 42, 95]. La evidencia inicial se obtiene mediante el conocimiento "en bruto" de la consistencia de tipos, luego la evidencia evoluciona a medida que se realizan las reducciones y derivaciones de un término, considerando la consistencia tanto de los predicados como de las funciones, junto con la consistencia de la transitividad de las evidencias. Una evidencia es un par de tipos, $\langle U_1, U_2 \rangle$, la cual nos permite soportar la consistencia de tipos [42] y refleja la información estática acerca de algunos términos. En este trabajo retomaremos la definición de evidencia de los trabajos de [11, 42], la cual no está definida en el trabajo de Toro y Tanter: $\varepsilon =_{def} \{\langle U_1, U_2 \rangle \mid U_i^{\,i \in 1,2} \in UTYPE\}$.

$$\frac{}{n^{Int} \in TERM_{\text{Int}}}\ (IU_n) \qquad \frac{}{b^{Bool} \in TERM_{Bool}}\ (IU_b) \qquad \frac{}{x^U \in TERM_U}\ (IU_x)$$

$$\frac{t^{U_1} \in TERM_{U_1} \qquad \varepsilon_1 \vdash U_1 \sim Int \qquad t^{U_2} \in TERM_{U_2} \qquad \varepsilon_2 \vdash U_2 \sim Int}{\varepsilon_1 t^{U_1} + \varepsilon_2 t^{U_2} \in TERM_{Int}}\ (IU_+)$$

$$\frac{\begin{array}{c} t^{U_1} \in TERM_{U_1} \qquad \varepsilon_1 \vdash U_1 \sim Bool \qquad U = (U_2 \sqcap U_3) \\ t^{U_2} \in TERM_{U_2} \qquad \varepsilon_2 \vdash U_2 \sim U \qquad t^{U_3} \in TERM_{U_3} \qquad \varepsilon_3 \vdash U_3 \sim U \end{array}}{\mathit{if}\ \varepsilon_1 t^{U_1}\ \mathit{then}\ \varepsilon_2 t^{U_2}\ \mathit{else}\ \varepsilon_3 t^{U_3} \in TERM_U}\ (IU_{if})$$

$$\frac{t^{U_1} \in TERM_{U_1} \qquad \varepsilon \vdash U_1 \sim U}{\varepsilon t^{U_1} :: U \in TERM_U}\ (IU_{::}) \qquad \frac{t^{U_2} \in TERM_{U_2}}{(\lambda x^{U_1}.\, t^{U_2}) \in TERM_{U_1 \to U_2}}\ (IU_\lambda)$$

$$\frac{t^{U_1} \in TERM_{U_1} \qquad \varepsilon_1 \vdash U_1 \sim U_{11} \to U_{12} \qquad t^{U_2} \in TERM_{U_2} \qquad \varepsilon_2 \vdash U_2 \sim U_{11}}{(\varepsilon_1 t^{U_1}) @^{U_{11} \to U_{12}} (\varepsilon_2 t^{U_2}) \in TERM_{U_{12}}}\ (IU_{App})$$

$$\frac{t_i^{U_i} \in TERM_{U_i} \qquad para\ cada\ i \in 1..n}{[\ell_i = t_i^{U_i\ i \in 1..n}] \in TERM_{[\ell_i : U_i\ i \in 1..n]}}\ (IU_{rec}) \qquad \frac{t^U \in TERM_U \qquad \varepsilon \vdash U \sim [\ell_i : U_i^{\ i \in 1..n}]}{\varepsilon t^U \bullet^{[\ell_i : U_i\ i \in 1..n]} \ell_j \in TERM_{U_j}}\ (IU_{proj})$$

Figura 4.8: Términos Intrínsecos Graduales de $\mathsf{GTFL}^{\oplus}_{rec}$.

La definición de los términos intrínsecos graduales t^U se expone en la Figura 4.8, utilizando reglas para enfatizar el uso de evidencias y la consistencia de sus respectivos juicios.

En el caso de la regla (IU_n) no es necesario tener una evidencia asociada a alguna instancia de tipo entero, porque la instancia en sí misma ya es de tipo entero, por ejemplo el número entero 4 no necesita tener asociada la evidencia de tipo entero, por lo que decimos que esa instancia pertenece al conjunto de términos de tipo entero, en este caso, $TERM_{Int}$. Este tipo de razonamiento también aplica a las constantes booleanas $false$ y $true$ (ver regla (IU_{Bool})); así como en el caso de la regla para variables (IU_x) donde x está definida con algún tipo U. En el caso de la regla (IU_+), las evidencias asociadas a cada uno de los operandos de la suma deben ser consistentes con el tipo entero, esto se ve reflejado en las premisas de la regla de tipificado $\varepsilon_1 \vdash U_1 \sim Int$ y $\varepsilon_1 \vdash U_2 \sim Int$. La forma en la que podemos leer este juicio $\varepsilon \vdash U_1 \sim U_2$ es la siguiente: *Dado un conjunto de evidencias se puede garantizar o se permite demostrar la relación de consistencia entre esos dos tipos.* Asimismo, es necesario que tanto el primer término de la suma como el segundo, pertenezcan al conjunto de términos $TERM_U$ es decir $t^{U_1} \in TERM_{U_1}$ y $t^{U_2} \in TERM_{U_2}$; de esta manera si se cumplen las cuatro premisas de la regla, entonces se podrá asegurar que la suma de ambos términos con sus respectivas evidencias también se encuentra definida en el conjunto $TERM_{Int}$. La regla (IU_{if}) nos dice en la premisa de la misma que t^{U_1} pertenece al conjunto $TERM_{U_1}$, así como los términos $t^{U_2} \in TERM_{U_2}$ y $t^{U_3} \in TERM_{U_3}$, y en particular la evidencia asociada al tipo de la *guardia* o de la condición booleana del if de tipo U_1 debe ser consistente con el tipo booleano, mientras que el tipo U_2 debe ser consistente con U (donde $U = U_2 \sqcap U_3$) a partir de la evidencia ε_2 (i.e. $\varepsilon_2 \vdash U_2 \sim U$), lo mencionado anteriormente aplica al tipo U_3 con su respectiva evidencia ε_3, y además debe de existir al menos una solución definida para el *meet* de los tipos U_2 y U_3 la cual debe ser de algún tipo en U, si todo lo anterior se cumple para esta regla entonces dadas las evidencias asociadas a cada uno de los términos intrínsecos de una expresión if permitirán asegurar que la expresión if también se encuentre dentro del conjunto de términos de tipo U, es decir, $TERM_U$. La regla asociada a la adscripción $(IU_{::})$ nos dice que el término intrínseco t^{U_1} asociado con una evidencia ε que pertenece al conjunto de términos intrínsecos $TERM_U$ deben de cumplir las siguientes premisas: $t^{U_1} \in TERM_{U_1}$, es decir, el término intrínseco con tipo U_1 debe pertenecer al conjunto de términos intrínsecos $TERM_{U_1}$ y la evidencia ε permitirá demostrar que la relación de consitencia entre U_1 y U se mantiene. La regla para la abstracción lambda (IU_λ) indica que este término debe pertenecer al conjunto de términos intrínsecos de tipo función $U_1 \to U_2$, para lo que se debe de cumplir en las premisas que el término t^{U_2} asociado al cuerpo de la función lambda pertenece al conjunto de términos intrínsecos $TERM_{U_2}$, es decir, al tipo del codominio de la función. La regla para la aplicación de función (IU_{app}) tiene como primer argumento de ésta un término intrínseco t el cual debe ser de tipo función $U_{11} \to U_{12}$, junto con una evidencia ε_1, esto es: $(\varepsilon_1 t^{U_1})$, y el segundo término intrínseco de la aplicación tendrá asociado otra evidencia $(\varepsilon_2 t^{U_2})$ y ésta deberá pertenecer al conjunto $TERM_{U_{12}}$, por lo que en las premisas se debe cumplir que tanto el término intrínseco t^{U_1} como t^{U_2}[16] pertenezcan a los conjuntos $TERM_{U_1}$ y $TERM_{U_2}$ respectivamente, a su vez que dada la evidencia ε_1 permita demostrar que la relación de consistencia entre U_1 y $U_{11} \to U_{12}$ se mantenga y que la evidencia ε_2 permita demostrar la consistencia entre U_2 y U_{11}, es decir, que el tipo del argumento de la aplicación sea consistente con el tipo del argumento que recibe la función en el primer argumento de la aplicación.

Las reglas de tipificación para los registros son (IU_{rec}) y su proyección (IU_{proj}), las cuales describen las relaciones consistentes de tipos entre los términos intrínsecos y las evidencias. Por un lado, la regla (IU_{rec}) contiene términos intrínsecos t_i cada uno en $TERM^{U_i}$ respectivamente. Por otro lado, la regla (IU_{proj}) necesita de una evidencia ε para representar la consistencia entre el tipo de un término intrínseco t^U y el tipo del registro $U \sim [\ell_i : U_i \ ^{i \in 1..n}]$ al recuperar el campo j-ésimo. Estas reglas expresan la idea de que cada término intrínseco mantiene su tipificado consistente en tiempo de ejecución [11, 42, 59], ya que la información de tipificación precisa se almacena en las deducciones de las evidencias.

A continuación se exponen algunos ejemplos usando derivaciones de tipo para términos intrínsecos graduales en $\mathsf{GTFL}^{\oplus}_{rec}$. Es importante destacar que en algunas partes de estos ejemplos no se utiliza la notación de pares asociada a las evidencias, por ejemplo cuando se tiene $\langle Int \rangle 1$ la evidencia del término 1 es justamente el par $\langle Int, Int \rangle$ el cual denotamos como $\langle Int \rangle$ solamente. Esto es debido a que en el levantamiento de la igualdad de tipos estáticos, ambos componentes de la evidencia siempre coinciden, por lo que las evidencias pueden

[16] Es importante mencionar que los términos t a los cuales se hace referencia en el texto son términos intrínsecos distintos, los cuales tienen cada uno un tipo asociado que los diferencía, así como sus respectivas evidencias.

representarse con un solo tipo gradual simplificado, aunque la forma de presentar la evidencia en general es en forma de pares [11].

Primer ejemplo usando (IU_{+})

$$\frac{\dfrac{}{2 \in TERM_{Int}}\,(IU_n) \qquad <Int>\vdash Int \sim Int \qquad \dfrac{}{3 \in TERM_{Int}}\,(IU_n) \qquad <Int>\vdash Int \sim Int}{(<Int>2+<Int>3) \in TERM_{Int}}\,(IU_{+})$$

Segundo ejemplo usando (IU_{if})

$$\dfrac{}{true \in TERM_{Bool}}\,(IU_b) \qquad \dfrac{}{1 \in TERM_{Int}}\,(IU_n) \qquad \dfrac{}{2 \in TERM_{Int}}\,(IU_n)$$

$$\frac{Int = (Int \sqcap Int) \qquad <Bool>\vdash Bool \sim Bool \qquad <Int>\vdash Int \sim Int \qquad <Int>\vdash Int \sim Int}{(if\ <Bool>true\ then\ <Int>1\ else\ <Int>2) \in TERM_{Int}}\,(IU_{if})$$

Tercer ejemplo usando (IU_{app}, IU_{λ}, IU_b, IU_{proj}, IU_{rec}, IU_n e IU_b)

$$\frac{\dfrac{}{1 \in TERM_{Int}}\,(IU_n) \qquad \dfrac{}{true \in TERM_{Bool}}\,(IU_b)}{[\ell_1 = 1, \ell_2 = true] \in TERM_{[\ell_1:Int,\ell_2:Bool]}}\,(IU_{rec})$$

$$\frac{\dfrac{<[\ell_1 : Int, \ell_2 : Bool]>\vdash [\ell_1 : Int, \ell_2 : Bool] \sim [\ell_1 : Int, \ell_2 : Bool]}{<[\ell_1 : Int, \ell_2 : Bool]>([\ell_1 = 1, \ell_2 = true] \bullet^{[\ell_1:Int,\ell_2:Bool]} \ell_2) \in TERM_{Bool}}\,(IU_{proj})}{(\lambda x : Int \oplus Bool.(<[\ell_1 : Int, \ell_2 : Bool]>[\ell_1 = 1, \ell_2 = true] \bullet^{[\ell_1:Int,\ell_2:Bool]} \ell_2)) \in TERM_{(Int \oplus Bool) \to Bool}}\,(IU_{\lambda})$$

$$<(Int \oplus Bool) \to Bool>\vdash (Int \oplus Bool) \to Bool \sim (Int \oplus Bool) \to Bool$$

$$\frac{<(Int \oplus Bool) \to Bool>\vdash Bool \sim (Int \oplus Bool) \qquad \dfrac{}{true \in TERM_{Bool}}\,(IU_b)}{\begin{array}{c}(<(Int \oplus Bool) \to Bool>(\lambda x : Int \oplus Bool.(<[\ell_1 : Int, \ell_2 : Bool]>[\ell_1 = 1, \ell_2 = true] \bullet^{[\ell_1:Int,\ell_2:Bool]} \ell_2)) \\ @^{(Int \oplus Bool) \to Bool} <Bool>true) \in TERM_{Bool}\end{array}}\,(IU_{app})$$

Cuarto ejemplo usando (IU_{app}, IU_{λ}, IU_x, IU_{proj}, IU_{rec}, IU_n e IU_b)

$$\frac{\dfrac{}{1 \in TERM_{Int}}\,(IU_n) \qquad \dfrac{}{true \in TERM_{Bool}}\,(IU_b)}{[\ell_1 = 1, \ell_2 = true] \in TERM_{[\ell_1:Int,\ell_2:Bool]}}\,(IU_{rec})$$

$$\frac{\dfrac{<[\ell_1 : Int, \ell_2 : Bool]>\vdash [\ell_1 : Int, \ell_2 : Bool] \sim [\ell_1 : Int, \ell_2 : Bool]}{<[\ell_1 : Int, \ell_2 : Bool]>([\ell_1 = 1, \ell = true] \bullet^{[\ell_1:Int,\ell_2:Bool]} \ell_2) \in TERM_{Bool}}\,(IU_{proj})}{(\lambda x : Int \oplus ?.(<[\ell_1 : Int, \ell_2 : Bool]>[\ell_1 = 1, \ell_2 = true] \bullet^{[\ell_1:Int,\ell_2:Bool]} \ell_2)) \in TERM_{(Int \oplus ?) \to Bool}}\,(IU_{\lambda})$$

$$<(Int \oplus ?) \to Bool>\vdash (Int \oplus ?) \to Bool \sim (Int \oplus ?) \to Bool \qquad <?>\vdash ? \sim (Int \oplus ?)$$

$$\frac{\dfrac{}{y \in TERM_{?}}\,(IU_x)}{\begin{array}{c}(<(Int \oplus ?) \to Bool>(\lambda x : Int \oplus ?.(<[\ell_1 : Int, \ell_2 : Bool]>[\ell_1 = 1, \ell_2 = true] \bullet^{[\ell_1:Int,\ell_2:Bool]} \ell_2))@^{(Int \oplus ?) \to Bool} \\ <?>y) \in TERM_{Bool}\end{array}}\,(IU_{app})$$

Figura 4.9: Ejemplos de derivaciones de tipos intrínsecos graduales de $\mathsf{GTFL}^{\oplus}_{rec}$.

Tanto la noción de reducción como las reglas de reducción de los términos intrínsecos se muestran en las Figuras 4.10 y 4.11, siguiendo [95] y [42]. Con los términos intrínsecos y el uso de las evidencias, las nociones de reducción se diseñan específicamente tanto para los *términos con evidencia* (et) como para los *valores con evidencia* (ev). La categoría de valores v tiene los mismos elementos que en los lenguajes anteriores, ahora categorizados en valores simples y valores con el tipo adscripción.

$$\varepsilon \in EVIDENCE, \quad et \in EvTERM, \quad ev \in EvVALUE, \quad t \in TERM_\star$$

$$\star \in UTYPE, \quad v \in VALUE, \quad u \in SimpleVALUE,$$

$$g \in EvFRAME, \quad f \in TmFRAME$$

$$\begin{array}{rcl}
et & ::= & \varepsilon t \\
ev & ::= & \varepsilon u \\
u & ::= & n \mid b \mid x \mid \lambda x : U.\, t \mid [\ell_i = u_i{}^{i\in 1..n}] \\
v & ::= & u \mid \varepsilon u :: U \\
g & ::= & \Box + et \mid ev + \Box \mid \mathit{if}\ \Box\ \mathit{then}\ et\ \mathit{else}\ et \mid \Box :: U \mid \Box @^U et \mid ev @^U \Box \mid \\
 & & [\ell_i = ev^{\,i\in 1..j+1}, \ell_j = \Box, \ldots, \ell_k = et_k^{k\in j+1..n}] \mid \Box \bullet^U \ell_j \\
f & ::= & g[\varepsilon \Box]
\end{array}$$

Noción de reducción:

$$\rightarrow : TERM_U \times (TERM_U \cup \{\mathbf{error}\})$$

$$\rightarrow_c : EvTERM \times (EvTERM \cup \{\mathbf{error}\})$$

$$\varepsilon n_1 + \varepsilon n_2 \rightarrow n_3 \qquad \text{donde } n_3 = n_1 [\![+]\!] n_2$$

$$\mathit{if}\ \varepsilon_1\, b\ \mathit{then}\ \varepsilon_2\, t^{U_2}\ \mathit{else}\ \varepsilon_3\, t^{U_3} \rightarrow \begin{cases} \varepsilon_2 t^{U_2} :: U_2 \sqcap U_3 & b = true \\ \varepsilon_3 t^{U_3} :: U_2 \sqcap U_3 & b = false \end{cases}$$

$$\varepsilon_1(\varepsilon_2 v :: U) \rightarrow_c \begin{cases} (\varepsilon_2 \circ^{=} \varepsilon_1) v \\ \mathbf{error} \quad \text{si } (\varepsilon_2 \circ^{=} \varepsilon_1) \text{ no está definido} \end{cases}$$

$$\varepsilon_1(\lambda x^{U_{11}}.t) @^{U_1 \to U_2}\ \varepsilon_2 u \rightarrow \begin{cases} \mathit{icod}(\varepsilon_1)(t[(\varepsilon_2 \circ^{=} \mathit{idom}(\varepsilon_1))u :: U_{11})/x^{U_{11}}]) :: U_2 \\ \mathbf{error} \quad \text{si no está definida} \end{cases}$$

$$\varepsilon[\ell_i = u_i{}^{i\in 1..n}] \bullet^{[\ell_i : U_i{}^{i\in 1..n}]} \ell_j \rightarrow \begin{cases} \mathit{iproj}(\varepsilon, \ell_j) u_j :: U_j \\ \mathbf{error} \quad \text{si j} \notin \text{1...n} \end{cases}$$

Figura 4.10: Sintaxis y noción de reducción en $\mathsf{GTFL}^{\oplus}_{rec}$.

Reglas de Reducción Intrínsecas: $\boxed{t \mapsto t}$

$$\mapsto : TERM_U \times (TERM_U \cup \{\mathbf{error}\})$$

$$\frac{t^U \to r \quad r \in (TERM_U \cup \{\mathbf{error}\})}{t^U \mapsto r}\ (R_{\to})$$

$$\frac{et \to_c et'}{g[et] \mapsto g[et']}\ (R_g) \qquad \frac{et \to_c \mathbf{error}}{g[et] \mapsto \mathbf{error}}\ (R_{g\,err})$$

$$\frac{t_1^U \mapsto t_2^U}{f[t_1^U] \mapsto f[t_2^U]}\ (R_f) \qquad \frac{t_1^U \mapsto \mathbf{error}}{f[t_1^U] \mapsto \mathbf{error}}\ (R_{f\,err})$$

Figura 4.11: Reglas de reducción intrínsecas en $\mathsf{GTFL}^{\oplus}_{rec}$.

Los valores evidencia son valores simples acompañados de evidencia εu y se utilizan en las reglas de reducción. Estas reglas, aseguran por un lado que, en tiempo de ejecución se siga manteniendo la propiedad de seguridad de lo contrario se envía un mensaje de error. Enfatizamos que cada regla de reducción $\to$ describe una reducción realizada en el sistema de tipificación gradual, donde se reduce un término. Y la regla de reducción $\to_c$ expresa la evaluación de un valor de adscripción donde la expresión ya está asignada a una etiqueta de tipo.

Lo descrito en los párrafos anteriores está basado en [42] donde la definición de transitividad consistente para un predicado de tipo P, como $\circ^P$ está definida por el marco de interpretación abstracta. El $\circ^=$ representa el *meet* de las evidencias [95]. Además, la consistencia de un término intrínseco se mantiene a través de los levantamientos, en particular, la consistencia de la igualdad levanta la igualdad de tipos estáticos por la información deducida de un juicio de tipo consistente. De manera formal se propone la siguiente definción[17] para $\circ^=$ la cual está basada en el trabajo desarrollado por Garcia et al. en [42]:

Sea P contenido en $GTYPE^2$ un predicado binario de tipos estáticos, (en este caso el predicado es la igualdad entre dos tipos, i.e. $=$), y supongamos:

$$\varepsilon_{12} \vdash P(U_1, U_2) \qquad y \qquad \varepsilon_{23} \vdash P(U_2, U_3)$$

.

Entonces podemos deducir una evidencia para la transitividad consistente como $(\varepsilon_{12} \circ^= \varepsilon_{23}) \vdash P(U_1, U_3)$ [18] donde $\circ^P$ es $\circ^= : GTYPE \times GTYPE \rightharpoonup GTYPE$ y está definido como:

$$\langle U_1, U_{21}\rangle \circ^= \langle U_{22}, U_3\rangle =_{def} \alpha^2(\{\langle U_1, U_3\rangle \in \gamma^2(U_1, U_3) \mid \exists\, U_2 \in \gamma(U_{21}) \cap \gamma(U_{22}).(U_1 = U_2) \wedge (U_2 = U_3)\})$$

Las funciones `idom`, `icod` y `iproj` utilizadas en la noción de reducción son el levantamiento de las funciones correspondientes extendidas para las evidencias (pares de tipos de unión gradual) siguiendo la Definición 4.5.9.

[17] La definición anterior no se encuentra en el trabajo de Toro y Tanter.
[18] Podemos reescribir $P(U_1, U_3)$ como $U_1 = U_3$.

$$
\begin{aligned}
idom(\langle U_1 \to U_2, U_1^{'} \to U_2^{'} \rangle) &= \langle U_1, U_1^{'} \rangle \\
idom(\varepsilon) &= \text{indefinido} \quad \text{en otro caso}
\end{aligned}
$$

$$
\begin{aligned}
icod(\langle U_1 \to U_2, U_1^{'} \to U_2^{'} \rangle) &= \langle U_2, U_2^{'} \rangle \\
icod(\varepsilon) &= \text{indefinido} \quad \text{en otro caso}
\end{aligned}
$$

$$
\begin{aligned}
iproj\left(\langle [\ell_i : {U_i}^{i\epsilon 1..n}], [\ell_i : {U'}_i^{i\epsilon 1..n}] \rangle, \ell_j\right) &= \langle U_j, U'_j \rangle \\
iproj\left(\varepsilon, \ell_j\right) &= \text{indefinido} \quad \text{en otro caso}
\end{aligned}
$$

La reducción intrínseca de términos (véase la Figura 4.11) incluye las reglas para evaluar términos en $\mathsf{GTFL}^{\oplus}_{rec}$ utilizando dos tipos de *frames*: marcos de evidencia g (*evidence frames*) y marcos de términos f (*term frames*). Los primeros son marcos para reducir los términos de evidencia y los segundos son marcos para enfatizar la reducción de un término de evidencia.

Propiedades

Consideremos un programa y dos versiones de éste: por un lado una versión con más tipos estáticos y, por otro lado, una versión con menos anotaciones de tipos. Un verificador convencional de tipos debe detectar estáticamente más errores de tipos que en la segunda versión. Sin embargo el resultado de ambos programas, en el mejor de los casos, debe ser el mismo.

Bajo un enfoque de lenguajes con tipificado gradual, la verificación de tales programas debe producir un resultado con el mismo tipo y eventualmente tener el mismo comportamiento en tiempo de ejecución. Es decir, se espera que ambas versiones tengan el mismo comportamiento porque de hecho son el mismo programa, excepto por los tipos que se obtienen mediante sus anotaciones de tipo explícitas, o bien debido a la interpretación gradual.

La **garantía gradual** (*gradual guarantee*) formaliza y garantiza esta relación. Si un programa con tipificado gradual está bien tipificado, incluso si todas las anotaciones de tipo son removidas, se debe garantizar que el programa sigue bien tipificado (*well-typed*).

Además, durante tiempo de ejecución, si un programa tipificado gradualmente se evalúa a un valor asociado a un tipo, entonces si se quitan algunas o todas las anotaciones de tipo en éste, el resultado de dicho programa debe ser un valor con un tipo equivalente [62]. La garantía gradual, definida por [82] expresa que incluso si en un programa se añaden o remueven anotaciones de tipo, no cambiará su comportamiento (semántica estática y dinámica). La garantía gradual relaciona el comportamiento de los programas que difieren solo con respecto a sus anotaciones de tipo. Es una relación formalizada a través de la precisión de términos y tipos.

El concepto relacionado con la garantía gradual se propone en [82] teniendo en cuenta tres propiedades:

1. Precisión del término (*term precision*).
2. Precisión del tipo (*type precision*).
3. Precisión con respecto a la evaluación (*precision with respect to evaluation*).

Otros autores utilizan las mismas propiedades y agregan la propiedad de seguridad de tipos (*type safety*) bajo el enfoque de tipificado gradual particularmente en el sistema de tipos propuesto [42, 51, 95].

En esta sección se muestran las demostraciones relacionadas con la garantía gradual estática, la propiedad de seguridad de tipos (*type safety property*) y la garantía gradual dinámica para $\mathsf{GTFL}^{\oplus}_{rec}$. Finalmente, se presenta el lenguaje intermedio mediante la inserción de conversiones (*cast insertions*).

Como los términos graduales utilizan evidencias ε, para refinar los juicios de tipos consistentes, las garantías graduales incluyen la noción de términos de precisión (*term precision*) como un parte fundamental para decidir esencialmente que cualquier programa que se ejecute sin errores continúa, incluso si tuviera menos términos precisos [42].

Por un lado, la garantía gradual estática (*static gradual guarantee*) sigue la idea de preservar la información estática lo más precisa posible, aún en el lenguaje con tipificado gradual [94]. Y por el otro lado, la garantía gradual dinámica (*dynamic gradual guarantee*) asegura que un programa tiene el mismo comportamiento en tiempo de ejecución, incluso si el mismo programa solo difiere en el tipificado. La reducción de estos programas deben mantener la precisión de términos y sus tipos [51].

Garantía Gradual Estática

Para probar la proposición de la Garantía Gradual Estática (*Static Gradual Guarantee*) 4.6.2, primero se tiene que probar la proposición que establece la equivalencia de términos con anotaciones completas (*fully-annotated terms*) 4.6.1 con respecto al lenguaje gradual, para después demostrar el Lema de Preservación de Precisión (*Lemma for Precision Preservation*) 4.6.1 sobre las derivaciones de tipos, así como el Lemma de Consistencia y Precisión en $UTYPE$ 4.6.2, el cual usa la definición de Precisión de Términos (*Term Precision*) 4.6.1 y la Precisión de Ambientes de Tipos (*Type Environment Precision*) 4.6.2.

Proposición 4.6.1 (Equivalencia de términos con anotaciones completas (*fully-annotated terms*))**.** *Para cualquier término totalmente tipificado (fully-annotated) $t \in TERM$, la derivación de tipo $\cdot \vdash_S t : T$ (en STFL_{rec}) se mantiene si y sólo si $\cdot \vdash t : T$ (en $\mathsf{GTFL}^{\oplus}_{rec}$) se mantiene.*

Demostración. La demostración se realiza por inducción sobre la derivación de tipos, la cual es directa debido a la concretización de tipos estática para cada tipo gradual único (*singleton*), incluyendo el caso para registros. ■

La precisión de términos está relacionada con un orden parcial de términos, y debe añadirse al criterio gradual. Esto no es fácil, sin embargo, permite establecer una relación entre dos términos y relacionarlos por su precisión de tipos, es decir, esta relación es monótona ya que mantiene la noción de tipos, incluyendo más o menos tipos desconocidos [107].

Definición 4.6.1 (Precisión de términos.)**.**

$$\frac{}{n \sqsubseteq n}\ (P_n) \qquad \frac{}{b \sqsubseteq b}\ (P_b) \qquad \frac{}{x \sqsubseteq x}\ (P_x) \qquad \frac{\widetilde{t_1} \sqsubseteq \widetilde{t_1'} \qquad \widetilde{t_2} \sqsubseteq \widetilde{t_2'}}{\widetilde{t_1} + \widetilde{t_2} \sqsubseteq \widetilde{t_1'} + \widetilde{t_2'}}\ (P_+)$$

$$\frac{\widetilde{t} \sqsubseteq \widetilde{t'} \qquad \widetilde{t_1} \sqsubseteq \widetilde{t_1'} \qquad \widetilde{t_2} \sqsubseteq \widetilde{t_2'}}{\mathit{if}\ \widetilde{t}\ \mathit{then}\ \widetilde{t_1}\ \mathit{else}\ \widetilde{t_2} \sqsubseteq \mathit{if}\ \widetilde{t'}\ \mathit{then}\ \widetilde{t_1'}\ \mathit{else}\ \widetilde{t_2'}}\ (P_{if}) \qquad \frac{\widetilde{t} \sqsubseteq \widetilde{t'} \qquad U \sqsubseteq U'}{\widetilde{t} :: U \sqsubseteq \widetilde{t'} :: U'}\ (P_{::})$$

$$\frac{\widetilde{t} \sqsubseteq \widetilde{t'} \qquad U_1 \sqsubseteq U_1'}{(\lambda x : U_1 . \widetilde{t}) \sqsubseteq (\lambda x : U_1' . \widetilde{t})}\ (P_\lambda) \qquad \frac{\widetilde{t_1} \sqsubseteq \widetilde{t_1'} \qquad \widetilde{t_2} \sqsubseteq \widetilde{t_2'}}{\widetilde{t_1}\ \widetilde{t_2} \sqsubseteq \widetilde{t_1'}\ \widetilde{t_2'}}\ (P_{app})$$

$$\frac{\widetilde{t_i} \sqsubseteq \widetilde{r_i} \qquad \mathit{para\ cada}\ i \in 1..n}{[\ell_i = \widetilde{t_i}^{\,i \in 1..n}] \sqsubseteq [\ell_i = \widetilde{r_i}^{\,i \in 1..n}]}\ (P_{rec}) \qquad \frac{\widetilde{t} \sqsubseteq \widetilde{t'}}{\widetilde{t}.\ell_j \sqsubseteq \widetilde{t'}.\ell_j \qquad \mathit{para\ alguna}\ j \in 1..n}\ (P_{proj})$$

Siguiendo la Definición 4.6.1, es necesario preservar la precisión de ambientes como sigue:

Definición 4.6.2 (Precisión de Ambientes de Tipo (*Type environment precision*))**.**

$$\frac{}{\cdot \sqsubseteq \cdot} \qquad \frac{\Gamma \sqsubseteq \Gamma' \qquad U \sqsubseteq U'}{\Gamma, x : U \sqsubseteq \Gamma', x : U'}$$

Ahora, es necesario demostrar el Lema de Preservación de Precisión 4.6.1 sobre las derivaciones de tipos.

Lema 4.6.1 (Preservación de Precisión (*Precision preservation*))**.** *Si $\Gamma \vdash \widetilde{t} : U$ y $\Gamma \sqsubseteq \Gamma'$, entonces $\Gamma' \vdash \widetilde{t} : U'$ para algún $U \sqsubseteq U'$.*

Demostración. Inducción en la derivación de tipos. La demostración se realiza para los casos de (U_{rec}) y (U_{proj}).

Caso (U_{rec})

Supóngase que $\Gamma \vdash [\ell_i = \widetilde{t}_i{}^{\,i\in 1..n}] : [\ell_i : U_i{}^{\,i\in 1..n}]$ y $\Gamma \sqsubseteq \Gamma'$.
Entonces, se quiere demostrar que $\Gamma' \vdash [\ell_i = \widetilde{t}_i{}^{\,i\in 1..n}] : [\ell_i : U_i{}^{\,i\in 1..n}]$, donde $[\ell_i : U_i{}^{\,i\in 1..n}] \sqsubseteq [\ell_i : U'_i{}^{\,i\in 1..n}]$.
Usando la inversión de una derivación de tipo para registros, se sabe que: $\Gamma \vdash \widetilde{t}_i : U_i$ para cada $i \in 1..n$.
Entonces, por Hipótesis de Inducción se tiene que: $\Gamma' \vdash \widetilde{t}_i : U'_i$ y $U_i \sqsubseteq U'_i$ para cada $i \in 1..n$. Usando la regla (U_{rec}) y la precisión de tipos para registros 4.5.4, el resultado se mantiene.

Caso (U_{proj})

Para probar $\Gamma' \vdash \widetilde{t}.\ell_j : U'_j$, donde $U_j \sqsubseteq U'_j$ para alguna $j \in 1..n$.
Considérese $\Gamma \vdash \widetilde{t}.\ell_j : U_j$ y $\Gamma \sqsubseteq \Gamma'$. Usando la inversión de una derivación de tipos para la proyección se obtiene: $\Gamma \vdash \widetilde{t} : [\ell_i : U_i{}^{\,i\in 1..n}]$, y por Hipótesis de Inducción, el siguiente juicio se mantiene:
$\Gamma' \vdash \widetilde{t} : [\ell_i : U'_i{}^{\,i\in 1..n}]$ donde $[\ell_i : U_i{}^{\,i\in 1..n}] \sqsubseteq [\ell_i : U'_i{}^{\,i\in 1..n}]$.
Entonces, por la relación de tipificado para la proyección de registros, se obtiene: $\Gamma' \vdash \widetilde{t}.\ell_j : U'_j$. Finalmente, el resultado se mantiene por la precisión de tipos de $U_i \sqsubseteq U'_i$ para cada $i \in 1..n$, en particular para ℓ_j, $U_j \sqsubseteq U'_j$.

■

Lema 4.6.2 (Consistencia y Precisión en $UTYPE$)**.** *Si $U_1 \sim U_2$ y $U_1 \sqsubseteq U'_1$ y $U_2 \sqsubseteq U'_2$ entonces $U'_1 \sim U'_2$.*

Demostración. La demostración es directa siguiendo las definiciones correspondientes. ■

La siguiente proposición 4.6.2 muestra la garantía gradual estáticamente:

Proposición 4.6.2 (Garantía Gradual Estática (*Static gradual guarantee*))**.** *Si $\cdot \vdash \widetilde{t_1} : U_1$ y $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$, entonces $\cdot \vdash \widetilde{t_2} : U_2$, para algún U_2 tal que $U_1 \sqsubseteq U_2$.*

Demostración. La demostración de la propiedad se realiza sobre términos abiertos, sin embargo basta con hacerlo para términos cerrados, i.e. si $\Gamma \vdash \widetilde{t_1} : U_1$ y $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$ entonces $\Gamma \vdash \widetilde{t_2} : U_2$ y $U_1 \sqsubseteq U_2$ [95].
La demostración se hace por inducción sobre la derivación de los tipos. Aquí se exponen los casos para registros, i.e. para las reglas U_{rec} y U_{proj}.

- **Caso** (U_{rec})
  Tomando $\widetilde{t_1} = [\ell_i = \widetilde{t_i}{}^{\,i\in 1..n}]$ con tipos $U_1 = [\ell_i : U_i{}^{\,i\in 1..n}]$ bajo Γ y considérese $\widetilde{t_2}$ tal que $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$.
  Se quiere probar que si $\Gamma \vdash \widetilde{t_2} : U_2$ entonces $U_1 \sqsubseteq U_2$.
  Por definición de precisión de términos, de $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$ se sabe que $\widetilde{t_2}$ debe tener la forma $\widetilde{t_2} = [\ell_i = \widetilde{r_i}{}^{\,i\in 1..n}]$ para algún término $\widetilde{r_i}$.

  Se sabe que $\Gamma \vdash \widetilde{t_i} : U_i{}^{\,i\in 1..n}$ para cada i por Regla de Inversión (U_{rec}). Usando la Hipótesis de Inducción sobre las premisas, se sabe: $\Gamma \vdash \widetilde{t_i} : U_i$ y $\widetilde{t_i} \sqsubseteq \widetilde{r_i}$ entonces $\Gamma \vdash \widetilde{r_i} : U'_i$ donde $U_i \sqsubseteq U'_i$.

  Por lo tanto, el término $\widetilde{t_2}$ tiene el tipo $U_2 = [\ell_i : U'_i{}^{\,i\in 1..n}]$ por la Regla (U_{rec}). De esto, se puede usar el Lema 4.6.1, con $\Gamma \sqsubseteq \Gamma$ para obtener $[\ell_i : U_i{}^{\,i\in 1..n}] \sqsubseteq [\ell_i : U'_i{}^{\,i\in 1..n}]$ y por lo tanto, el resultado se mantiene.

- **Caso** (U_{proj})
  Tomando $\widetilde{t_1} = \widetilde{t}.\ell_j$ con el tipo U_j para algún $j \in 1..n$ bajo Γ, y considerando $\widetilde{t_2}$ tal que $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$. Se quiere probar que si $\Gamma \vdash \widetilde{t_2} : U_2$ entonces $U_j \sqsubseteq U_2$.
  Por la regla de tipificado de inversión (U_{proj}) para $\widetilde{t_1}$, se sabe que $\Gamma \vdash \widetilde{t} : U_1$ donde $U_1 = [\ell_i : U_i{}^{\,i\in 1..n}]$.
  Usando la Hipótesis de Inducción en esta premisa, se sabe que $\Gamma \vdash \widetilde{t'} : U'_1$ donde $U_1 \sqsubseteq U'_1$. Y por la regla (U_{proj}) $\Gamma \vdash \widetilde{t'}.\ell_j : \widetilde{proj}(U'_1, \ell_j)$.

  Usando la definición de precisión de términos, para $\widetilde{t_1} \sqsubseteq \widetilde{t_2}$, se sabe que $\widetilde{t_2}$ debe tener la forma $\widetilde{t_2} = \widetilde{t'}.\ell_j$ y por lo tanto $\widetilde{t} \sqsubseteq \widetilde{t'}$.

  De esto, se puede usar el Lema 4.6.1 para obtener $U_j \sqsubseteq U'_j$, y por lo tanto, el resultado se mantiene.

■

Seguridad de Tipos (*type safety*) .

Una propiedad fundamental de cualquier lenguaje tipificado es la seguridad de tipos. En un lenguaje gradual esta propiedad se adapta para asegurar que si un programa gradual termina, entonces el valor resultante está bien definido, o bien, éste puede ser reducido a otro término gradual. En otro caso, el programa gradual puede terminar con un error debido a la conversión de tipos, i.e. un *cast error* [42].

Para estudiar la última propiedad de la seguridad de tipos 4.6.6, las Formas Canónicas se exponen a continuación en 4.6.3, la propiedad de Sustitución de valores 4.6.3 junto con las demostraciones de las reducción bien definidas $\rightarrow$ y $\mapsto$ dadas en la Figura 4.10 y 4.11.

Proposición 4.6.3 (Formas Canónicas en $\mathsf{GTFL}^{\oplus}_{rec}$)**.** *Consideremos un valor* $v \in TERM_U$. *Entonces,* $v = u$, *o* $v = \varepsilon\, u :: U$ *con* $u \in TERM_{U'}$ *y* $\varepsilon \vdash U' \sim U$. *Además:*

1. *Si* $U = Int$ *entonces* $v = n$ *o* $v = \varepsilon\, n :: Int$ *con* $n \in TERM_{Int}$.
2. *Si* $U = Bool$ *entonces* $v = b$ *o* $v = \varepsilon\, b :: Bool$ *con* $b \in TERM_{Bool}$.
3. *Si* $U = U_1 \rightarrow U_2$ *entonces* $v = (\lambda x^{U_1}.t^{U_2})$ *con* $t^{U_2} \in TERM_{U_2}$ *o* $v = \varepsilon(\lambda x^{U_1'}.t^{U_2'}) :: U_1 \rightarrow U_2$ *con* $t^{U_2'} \in TERM_{U_2'}$ *y* $\varepsilon \vdash U_1' \rightarrow U_1' \sim U_1 \rightarrow U_2$.
4. *Si* $U = [\ell_j : {U_i}^{i\in 1..n}]$ *entonces* $v = [\ell_i = {v_i}^{i\in 1..n}]$ *o* $v = \varepsilon\big([\ell_i = {v_i}^{i\in 1..n}]\big) :: [\ell_i : {U_i}^{i\in 1..n}]$ *con* $[\ell_i = {v_i}^{i\in 1..n}] \in TERM_{U'}$ *y* $\varepsilon \vdash U' \sim [\ell_i : {U_i}^{i\in 1..n}]$.

Para probar que el lenguaje gradual es seguro, el Lema de Sustitución 4.6.3 está dado para la aplicación de términos:

Lema 4.6.3 (Sustitución)**.** *Si* $t^U \in TERM_U$ *y* $v \in TERM_{U_1}$, *entonces* $t^U[v/x^{U_1}] \in TERM_U$.

Demostración. La demostración es directa y se lleva a cabo por inducción sobre la derivación de t^U. ∎

Para probar la seguridad de un lenguaje gradual, es necesario demostrar que están bien definidas las nociones de reducción expuestas en la Figura 4.10 por medio de la siguientes proposiciones 4.6.4 y 4.6.5. Ambas proposiciones consideran a $CONFIG_U$ como el conjunto de términos de un tipo esperado, representados por el término de adscripción correspondiente; si la consistencia de tipos no se alcanza en este punto, entonces se utiliza el caso **error** para regresar uno en tiempo de ejecución.

La primera proposición usa la definición de términos intrínsecos graduales dada en la Figura 4.8, ya que cada término intrínseco t^U representa una derivación de tipificación del tipo correspondiente U.

Proposición 4.6.4 ($\rightarrow$ está bien definida)**.** *Si* $t^U \rightarrow r$, *entonces* $r \in CONFIG_U \cup \{\mathbf{error}\}$.

Demostración. La demostración se realiza por inducción sobre la estructura de la derivación de $t^U \rightarrow r$, considerando la última regla usada en la derivación. El caso de la proyección para registros está dado aquí.
Supóngase que $t^U = \varepsilon[\ell_i = {u_i}^{i\in 1..n}].\ell_j$ y debido a la definición de (IU_{proj}) en la Figura 4.8, $([\ell_i = {u_i}^{i\in 1..n}]) \in TERM_{U'}$ se mantiene. Entonces, por la regla (IU_{rec}), $U' = [\ell_i : {U'}_i^{\,i\in 1..n}]$ donde cada valor $u_i^{U_i'} \in TERM_{U_i'}$.
Por lo tanto, la evidencia en t^U es tal que $U' \sim [\ell_i : {U_i}^{i\in 1..n}]$ se mantiene.
Por lo tanto, la reducción $\varepsilon[\ell_i = {u_i}^{i\in 1..n}].\ell_j \rightarrow iproj(\varepsilon, \ell_j)u_j :: U$ se realiza y el término resultante es una proyección $iproj(\varepsilon, \ell_j)u_j :: U \in TERM_U$ el cual se mantiene por la regla ($\mathrm{IU}_{::}$) como $iproj(\varepsilon, \ell_j) \vdash U_j' \sim U$. ∎

Proposición 4.6.5 ($\mapsto$ está bien definida)**.** *Si* $t^U \mapsto r$, *then* $r \in CONFIG_U \cup \{\mathbf{error}\}$.

Demostración. La demostración se realiza por inducción en la estructura de la derivación de $t^U \mapsto r$. La demostración es la misma que la descrita en [95] debido a que ninguna regla nueva es añadida. ∎

Ahora, se prueba la seguridad de tipos de un lenguaje gradual, siguiendo la Proposición 4.6.6, para los casos de registros de las reglas (IU_{rec}) y (IU_{proj}).

Proposición 4.6.6 (Seguridad de Tipos (*Type Safety*)). *Si $t^U \in TERM_U$, entonces t^U es un valor v; $t^U \mapsto t'^U$ para algún término $t'^U \in TERM_U$ o $t^U \mapsto$ **error**;*

Demostración. La demostración se desarrolla por inducción sobre la estructura de t^U.

- **Caso ($\mathbf{IU}_{rec}$)**
  Tomando $t^U = [\ell_i = t_i^{U_i\ i\in 1..n}]$ cuya definición de (IU_{rec}), $U = [\ell_i : U_i\ ^{i\in 1..n}]$ y $t_i^{U_i} \in TERM_{U_i}$ para alguna $i \in 1..n$. La demostración procede por análisis de casos de la Hipótesis de Inducción de los términos $t_i^{U_i}$:

  1. Los términos t_i son valores simples u_i, por lo tanto t^U es un valor registo (*value record*), i.e., $t^U = [\ell_i = u_i^{U_i\ i\in 1..n}]$.
  2. Algunos de los términos t_i son valores adscritos (*ascribed values*), sin pérdida de generalidad consideremos que los primeros $j+1$ campos son tales valores:
     $[\ell_i = v\ ^{i\in 1..j+1}, \ell_j = \varepsilon t_j, \ldots, \ell_k = \varepsilon t_k^{k\in j+1..n}]$.
     a) Si un marco de evidencias (*evidence frames*) en el campo ℓ_j permite la reducción de t en tal campo, entonces t_j es reducido por la regla (R_g) y existe un t'_j o por la Regla ($R_{g\,err}$) la reducción $t_j \mapsto$ **error** se realiza. Por lo tanto, la reducción de t_j pertenece a $CONFIG_U \cup \{\mathbf{error}\}$.
     b) Si un marco de términos (*terms frame*) en el campo ℓ_j mantiene la reducción para algún $t_j \in TERM_{U_i} \cup \{\mathbf{error}\}$ entonces la reducción de t se desarrolla por la Regla (R_f) o por la Regla ($R_{f\,err}$).

     En ambos casos la Proposición 4.6.5 se mantiene.

- **Caso ($\mathbf{IU}_{proj}$)**
  Considérese $t^U = \varepsilon(t'^{U'}.\ell_j)$ y por la Regla (IU_{proj}) el término t' pertenece a $TERM_{U'}$ y
  $\varepsilon \vdash U' \sim [\ell_i : U_i\ ^{i\in 1..n}]$ se mantiene. Por Hipótesis de Inducción sobre $t'^{U'}$, uno de los siguientes puntos se mantiene:

  1. $t'^{U'}$ es un valor simple, por lo tanto, por las Formas Conónicas 4.6.3 es igual a $[\ell_i = v_i\ ^{i\in 1..n}]$. Por lo tanto, por la regla ($R_{\rightarrow}$), $t^U \mapsto r$ se realiza y $r = iproj(\varepsilon, \ell_j)u_j :: U$, por la Proposición 4.6.4 el resultado se mantiene.
  2. $t'^{U'}$ es un valor adscrito v, entonces $t^U \mapsto r$ para alguna $r \in CONFIG_U \cup \{error\}$, por la regla ($R_g$) o la regla ($R_g\,err$) se mantiene por la Proposición 4.6.5.
  3. $t'^{U'}$ es un término tal que $t'^{U'} \mapsto r'$ y $r' \in CONFIG_{U'} \cup \{error\}$. Por lo tanto, $t^U \mapsto r$ con $r \in CONFIG_U \cup \{error\}$ lo cual se mantiene por la Proposición 4.6.5 y una de las siguientes reglas (R_f) o ($R_{f\,err}$).

■

Garantía Gradual Dinámica

En esta sección, se expone la prueba formal de la Garantía Gradual Dinámica para $\mathsf{GTFL}^{\oplus}_{rec}$. Esta propiedad en cualquier Lenguaje con Tipificado Gradual expresa que si tenemos dos programas que contienen el mismo código, con la única diferencia de que uno tiene anotaciones de tipo explícitamente, y el otro no necesariamente presenta todas las anotaciones de tipo, entonces decimos que ambos programas son el mismo ya que su comportamiento no debe cambiar dinámicamente (pues de hecho, es el mismo programa). Por lo tanto, el programador puede estar seguro de que incluso si elimina las anotaciones de tipos en su programa, éste sigue bien tipificado (sin insertar conversiones de tipo *cast*), por lo que el comportamiento y el tipo esperado del programa en tiempo de ejecución se conserva [82].

La garantía gradual dinámica expresa la idea de que los programas se ejecutan al menos tan bien como sus contrapartes más estáticas. Y los programas más estáticos siguen ejecutándose igual de mal como se venían ejecutando desde un inicio. En otras palabras, añadir más tipos estáticos no arregla los errores de tipo anteriores. [42] expone que un lenguaje gradual debe ser seguro aun si no hay forma en que se puedan seguir evaluando (*do not get stuck*). A su vez se incluye el caso cuando éstos puedan terminar con errores asociados a

la de conversión de tipos (*cast*). Siguiendo esta idea y el enfoque de AGT permite satisfacer este criterio por construcción. En las siguientes secciones y subsecciones se muestra que un lenguaje con registros mantiene esta propiedad.

Para probar la última garantía, la precisión de los términos intrínsecos se debe preservar en tiempo de ejecución. Por lo tanto, el primer paso es establecer la precisión de términos no cerrados, definiendo la relación $\Omega \vdash t^U \sqsubseteq r^{U'}$ entre dos términos, donde Ω denota la relación de precisión entre dos variables y sus tipos, los cuales deben de ser consistentes.

Definición 4.6.3 (Precisión de términos intrínsecos)**.** *Sea* $\Omega \in P\left(VAR_\star \times VAR_\star\right)$ *definido como* $\Omega ::= \{x^{U_{i1}} \sqsubseteq x^{U_{i2}}\}$*. Definimos la relación de orden* $(\cdot \vdash \cdot \sqsubseteq \cdot) \in P\left(VAR_\star \times VAR_\star\right) \times TERM_\star \times TERM_\star$*, definido inductivamente como* [19]*:*

$$\frac{}{\Omega \vdash n \sqsubseteq n}\ (ITP_n) \qquad \frac{}{\Omega \vdash b \sqsubseteq b}\ (ITP_b) \qquad \frac{}{\Omega \cup \{x^{U_1} \sqsubseteq x^{U_2}\} \vdash x^{U_1} \sqsubseteq x^{U_2}}\ (ITP_x)$$

$$\frac{\Omega \vdash t^{U_{11}} \sqsubseteq t^{U_{21}} \qquad \varepsilon_{11} \sqsubseteq \varepsilon_{21} \qquad \Omega \vdash t^{U_{12}} \sqsubseteq t^{U_{22}} \qquad \varepsilon_{12} \sqsubseteq \varepsilon_{22}}{\Omega \vdash \left(\varepsilon_{11} t^{U_{11}} + \varepsilon_{12} t^{U_{12}}\right) \sqsubseteq \left(\varepsilon_{21} t^{U_{21}} + \varepsilon_{22} t^{U_{22}}\right)}\ (ITP_+)$$

$$\frac{\begin{array}{cc}\Omega \vdash t^{U_{11}} \sqsubseteq t^{U_{21}} & \varepsilon_{11} \sqsubseteq \varepsilon_{21} \\ \Omega \vdash t^{U_{12}} \sqsubseteq t^{U_{22}} & \varepsilon_{12} \sqsubseteq \varepsilon_{22} \\ \Omega \vdash t^{U_{13}} \sqsubseteq t^{U_{23}} & \varepsilon_{13} \sqsubseteq \varepsilon_{23}\end{array}}{\Omega \vdash \mathit{if}\ \varepsilon_{11} t^{U_{11}}\ \mathit{then}\ \varepsilon_{12} t^{U_{12}}\ \mathit{else}\ \varepsilon_{13} t^{U_{13}} \sqsubseteq \mathit{if}\ \varepsilon_{21} t^{U_{21}}\ \mathit{then}\ \varepsilon_{22} t^{U_{22}}\ \mathit{else}\ \varepsilon_{23} t^{U_{23}}}\ (ITP_{if})$$

$$\frac{\Omega \vdash t^{U_{11}} \sqsubseteq t^{U_{21}} \qquad U_{12} \sqsubseteq U_{22} \qquad \varepsilon_1 \sqsubseteq \varepsilon_2}{\left(\varepsilon_1\, t^{U_{11}} :: U_{12}\right) \sqsubseteq \left(\varepsilon_2\, t^{U_{21}} :: U_{22}\right)}\ (ITP_{::}) \qquad \frac{U_{11} \sqsubseteq U_{21} \qquad \Omega \cup \{x^{U_{11}} \sqsubseteq x^{U_{21}}\} \vdash t^{U_{12}} \sqsubseteq t^{U_{22}}}{\Omega \vdash \left(\lambda x^{U_{11}}.\, t^{U_{12}}\right) \sqsubseteq \left(\lambda x^{U_{21}}.\, t^{U_{22}}\right)}\ (ITP_\lambda)$$

$$\frac{\Omega \vdash t^{U_{11}} \sqsubseteq t^{U_{21}} \qquad \varepsilon_{11} \sqsubseteq \varepsilon_{21} \qquad \Omega \vdash t^{U_{12}} \sqsubseteq t^{U_{22}} \qquad \varepsilon_{12} \sqsubseteq \varepsilon_{22} \qquad U_1 \sqsubseteq U_2}{\Omega \vdash \varepsilon_{11}^{U_{11}} @^{U_1} \varepsilon_{12}^{U_{12}} \sqsubseteq \varepsilon_{21}^{U_{21}} @^{U_2} \varepsilon_{22}^{U_{22}}}\ (ITP_{app})$$

$$\frac{\Omega \vdash t_i^{U_{1i}} \sqsubseteq r_i^{U_{2i}} \qquad \textit{para cada } i \in 1..n}{\Omega \vdash \left[\ell_i = t_i^{U_{1i}}{}^{\, i \in 1..n}\right] \sqsubseteq \left[\ell_i = r_i^{U_{2i}}{}^{\, i \in 1..n}\right]}\ (ITP_{rec})$$

$$\frac{\Omega \vdash t_1^{U_1} \sqsubseteq t_2^{U_2} \qquad \varepsilon_1 \sqsubseteq \varepsilon_2}{\Omega \vdash \varepsilon_1 t_1^{U_1} \bullet^{[\ell_{i1}:U_{i1}\, i \in 1..n]} \ell_j \sqsubseteq \varepsilon_2 t_2^{U_2} \bullet^{[\ell_{i2}:U_{i2}\, i \in 1..n]} \ell_j}\ (ITP_{proj})$$

Definición 4.6.4 (*Well Formedness* Ω.)**.** *Decimos que* Ω *se encuentra bien formada si y sólo para toda* $\{x^{G_{i1}} \sqsubseteq x^{G_{i2}}\}$ *en* Ω *entonces* $G_{i1} \sqsubseteq G_{i2}$*.*

Proposición 4.6.7 (Precisión de términos intrínsecos con Ω)**.** *Si* $\Omega \vdash t^{U_1} \sqsubseteq t^{U_2}$ *para alguna* $\Omega \in P\left(VAR_\star \times VAR_\star\right)$*, entonces* $U_1 \sqsubseteq U_2$*.*

Demostración. La demostración se realiza por inducción sobre la derivación de $\Omega \vdash t^{U_1} \sqsubseteq t^{U_2}$. Se demuestran los casos para registros, esto es para las reglas (ITP_{rec}) y (ITP_{proj}).

- **Caso** (ITP_{rec})
  Considérese $t^{U_1} = \left[\ell_i = t_i^{U_{1i}\, i \in 1..n}\right]$ y $t^{U_2} = \left[\ell_i = t_i^{U_{2i}\, i \in 1..n}\right]$.
  Entonces por Hipótesis de Inducción, si $\Omega \vdash t_i^{U_{1i}} \sqsubseteq t_i^{U_{2i}}$ entonces $U_{1i} \sqsubseteq U_{2i}$ para cada $i \in 1..n$. Y usando estas hipótesis en las reglas correspondientes a la precisión de tipos en registros (*record type precision*) (Proposición 4.5.4) el resultado se mantiene.

[19] Recordemos que $VAR_\star$ y $TERM_\star$ hacen referencia a variables intrínsecas y términos intrínsecos respectivamente.

- **Caso ($\mathbf{ITP}_{proj}$)**
  Si $t^{U_1} = \varepsilon_1 r^{U_1'}.\ell_j$ y $t^{U_2} = \varepsilon_2 r^{U_2'}.\ell_j$ tal que $\Omega \vdash t^{U_1} \sqsubseteq t^{U_2}$, y entonces por Hipótesis de Inducción $\Omega \vdash r^{U_1'} \sqsubseteq r^{U_2'}$ entonces $U_1' \sqsubseteq U_2'$. Por lo tanto, usando la definición de precisión de tipos, Proposición 4.5.4, cada tipo en los campos en $r^{U_1'}$ y $r^{U_2'}$ preserva la consistencia de tipos. Por lo tanto, en el resultado se mantiene la precisión como en el campo j-ésimo $U_{1j} \sqsubseteq U_{2j}$.

■

Proposición 4.6.8 (Marcos y evidencias)**.** *Sea $g_1, g_2 \in EvFRAME$ tal que $g_1[\varepsilon_{11} t_1^{U_1}] \in TERM_{U_1'}$ y $g_2[\varepsilon_{21} t_1^{U_2}] \in TERM_{U_2'}$ con $U_1' \sqsubseteq U_2'$.*
Entonces, si $g_1[\varepsilon_{11} t_1^{U_1}] \sqsubseteq g_2[\varepsilon_{21} t_1^{U_2}]$, $\varepsilon_{12} \sqsubseteq \varepsilon_{22}$ y $t_2^{U_1} \sqsubseteq t_2^{U_2}$, entonces $g_1[\varepsilon_{12} t_2^{U_1}] \sqsubseteq g_2[\varepsilon_{22} t_2^{U_2}]$.

Demostración. La demostración se realiza usando análisis de casos sobre g_i y la precisión de términos intrínsecos.

- **Caso** ($[\ell_i = \varepsilon_i v_i{}^{\,i \in 1..j+1}, \ell_j = \square, \ldots, \ell_k = \varepsilon_k t_k{}^{\,k \in j+1..n}]$)
  Considérese g_i, para $i \in \{1,2\}$, tiene la forma $[\ell_1 = \varepsilon_1 v_1, \ldots, \ell_j = \square, \ldots, \ell_k = t^{U_{1k}\ k \in j+1..n}]$ y un término t_1 tal que $g_1[\varepsilon_{11} t_1^{U_1}] \sqsubseteq g_2[\varepsilon_{21} t_1^{U_2}]$ lo cual por la regla de precisión de términos intrínsecos ITP_{rec}, $r_i^{U_{1i}} \sqsubseteq r_i^{U_{2i}}$ se mantiene para cada r_i con $i \in 1..n$, los cuales representan la relación de precisión punto a punto de los valores de evidencias o bien son los términos de evidencia en los correspondientes registros. Ahora se toma $\varepsilon_{12} \sqsubseteq \varepsilon_{22}$ y $t_2^{U_1} \sqsubseteq t_2^{U_2}$, para llenarse en los *frames* g_1 y g_2. Entonces, los términos $g_1[\varepsilon_{12} t_2^{U_1}]$ y $g_2[\varepsilon_{22} t_2^{U_2}]$ pueden estar relacionados a través de la precisión de términos intrínsecos usando la relación de precisión en cada campo, y la última relación de precisión de t_2

- **Caso** ($\square.\ell_j$)
  Entonces para $i \in \{1,2\}$, g_i debe tener la forma $\square.\ell_j$ y $g_1[\varepsilon_{11} t_1^{U_1}] \in TERM_{U_1'}$, $g_2[\varepsilon_{21} t_1^{U_2}] \in TERM_{U_2'}$ con $U_1' \sqsubseteq U_2'$. Considérese $g_1[\varepsilon_{11} t_1^{U_1}] \sqsubseteq g_2[\varepsilon_{21} t_1^{U_2}]$, entonces por las premisas de la precisión de términos intrínsecos ITP_{proj}, $t_1^{U_1} \sqsubseteq t_1^{U_2}$ y $\varepsilon_{11} \sqsubseteq \varepsilon_{21}$ se mantiene.
  Ahora supongamos que $\varepsilon_{12} \sqsubseteq \varepsilon_{22}$ y $t_2^{U_1} \sqsubseteq t_2^{U_2}$ entonces $\varepsilon_{12}\big(t_2^{U_1} \bullet^{[\ell_i : U_i\ i \in 1..n]} \ell_j\big) \sqsubseteq \varepsilon_{22}\big(t_2^{U_2} \bullet^{[\ell_i : U_i\ i \in 1..n]} \ell_j\big)$ puede derivarse y el resultado se mantiene.

■

Proposición 4.6.9 (Marcos y evidencias, 2)**.** *Sea $g_1, g_2 \in EvFRAME$ tal que $g_1[\varepsilon_1 t^{U_1}] \in TERM_{U_1'}$ y $g_2[\varepsilon_2 t^{U_2}] \in TERM_{U_2'}$ con $U_1' \sqsubseteq U_2'$. Entonces, si $g_1[\varepsilon_1 t^{U_1}] \sqsubseteq g_2[\varepsilon_2 t^{U_2}]$, entonces $t^{U_1} \sqsubseteq t^{U_2}$, y $\varepsilon_1 \sqsubseteq \varepsilon_2$.*

Demostración. La demostración se realiza por análisis de casos sobre g_i y precisión de términos intrínsecos.

- **Caso** $[\ell_i = v_i{}^{\,i \in 1..n}, \ell_j = \square, \ldots, \ell_k = t_k{}^{\,k \in j+1..n}]$
  Considérese g_i, para $i \in \{1,2\}$, con la forma $[\ell_1 = \varepsilon_1 v_1, \ldots, \ell_j = \square, \ldots, \ell_k = t^{U_{1k}\ k \in j+1..n}]$ y t tal que $g_1[\varepsilon_1 t^{U_1}] \sqsubseteq g_2[\varepsilon_2 t^{U_2}]$, y $U_1' \sqsubseteq U_2'$ que es $[\ell_i : U_{1_i}{}^{\,i \in 1..n}] \sqsubseteq [\ell_i : U_{2_i}{}^{\,i \in 1..n}]$. Entonces, por la regla (ITP_{rec}), los pares en los registros correspondientes son $r_i^{U_{1i}} \sqsubseteq r_i^{U_{2i}}$ para valores evidencia (*evidence values*) y términos evidencia (*evidence terms*) en cada campo se mantiene. En particular, el campo j-ésimo con $t^{U_{1j}} \sqsubseteq t^{U_{2j}}$ y $\varepsilon_1 \sqsubseteq \varepsilon_2$.

- **Caso** ($\square.\ell_j$)
  Considérese $g_1[\varepsilon_1 t^{U_1}] = \varepsilon_1 t^{U_1} \bullet^{[\ell_i : U_i\ i \in 1..n]} \ell_j$ y $g_2[\varepsilon_2 t^{U_2}] = \varepsilon_2 t^{U_2} \bullet^{[\ell_i : U_i\ i \in 1..n]} \ell_j$ tal que $U_1' \sqsubseteq U_2'$.
  Supóngase que $\varepsilon_1\big(t^{U_1} \bullet^{[\ell_i : U_i\ i \in 1..n]} \ell_j \sqsubseteq \big)\varepsilon_2\big(t^{U_2} \bullet^{[\ell_i : U_i\ i \in 1..n]} \ell_j\big)$. Por la regla ($ITP_{proj}$), las relaciones $t^{U_1} \sqsubseteq t^{U_2}$ y $\varepsilon_1 \sqsubseteq \varepsilon_2$ garantizan que el resultado se mantiene.

■

Proposición 4.6.10. *Sea $f_1, f_2 \in TmFRAME$ tal que $f_1[\varepsilon_1 t_1^{U_1}] \in TERM_{U_1'}$, $f_2[\varepsilon_2 t_1^{U_2}] \in TERM_{U_2'}$, con $U_1' \sqsubseteq U_2'$. Entonces, si $f_1[t_1^{U_1}] \sqsubseteq f_2[t_1^{U_2}]$ y $t_1^{U_1} \sqsubseteq t_2^{U_2}$ con $f_1[t_1^{U_1}] \sqsubseteq f_2[t_1^{U_2}]$.*

Demostración. La demostración es la misma que en [95]. ■

Proposición 4.6.11. *Sea* $f_1, f_2 \in TmFRAME$ *tal que* $f_1[\varepsilon_1 t^{U_1}] \in TERM_{U_1'}$, $f_2[\varepsilon_2 t^{U_2}] \in TERM_{U_2'}$, *con* $U_1' \sqsubseteq U_2'$. *Entonces, si* $f_1[t^{U_1}] \sqsubseteq f_2[t^{U_2}]$ *y* $t^{U_1} \sqsubseteq t^{U_2}$.

Demostración. La demostración es la misma que en [95]. ∎

Proposición 4.6.12 (La sustitución preserva la precisión.)**.** *Si* $\Omega \cup \{x^{U_3} \sqsubseteq x^{U_4}\} \vdash t^{U_1} \sqsubseteq t^{U_2}$ *y* $\Omega \vdash r^{U_3} \sqsubseteq r^{U_4}$, *entonces* $\Omega \vdash [r^{U_3}/x^{U_3}]t^{U_1} \sqsubseteq [r^{U_4}/x^{U_4}]t^{U_2}$.

Demostración. La demostración se realiza por inducción sobre la derivación de $t^{U_1} \sqsubseteq t^{U_2}$, que es un análisis de casos de la última regla utilizada en la derivación. Todos los casos pueden ser los triviales (no tienen premisas) o bien se siguen por la hipótesis de inducción [95].

- **Caso** (ITP_{rec})
  Sabemos que $t = [\ell_i = t_i^{i \in 1..n}]$ y $t^{U_1} \sqsubseteq t^{U_2}$. Considérese $\Omega \cup \{x^{U_3} \sqsubseteq x^{U_4}\} \vdash t^{U_1} \sqsubseteq t^{U_2}$ y $\Omega \vdash r^{U_3} \sqsubseteq r^{U_4}$.
  Por Hipótesis de Inducción sobre cada campo, la sustitución se mantiene:
  $\Omega \vdash [\ell_i = t_i^{i \in 1..n}]^{U_1}[r^{U_3}/x^{U_3}] \sqsubseteq [\ell_i = t_i^{i \in 1..n}]^{U_2}[r^{U_4}/x^{U_4}]$.
  Y usando éstas como hipótesis en la regla (ITP_{rec}) y aplicando la definición de sustitución sobre registros, el resultado se mantiene.

- **Caso** (ITP_{proj})
  Se sabe que $t = [\ell_i = t_i^{i \in 1..n}].\ell_j$ y considérese $\Omega \cup \{x^{U_3} \sqsubseteq x^{U_4}\} \vdash t^{U_1} \sqsubseteq t^{U_2}$ y $\Omega \vdash r^{U_3} \sqsubseteq r^{U_4}$.
  La demostración se dirige de la siguiente forma:
  $\Omega \vdash \big([\ell_i = t_i^{i \in 1..n}].\ell_j^{U_1}\big)[r^{U_3}/x^{U_3}] \sqsubseteq \big([\ell_i = t_i^{i \in 1..n}].\ell_j^{U_2}\big)[r^{U_4}/x^{U_4}]$.
  Por Hipótesis de Inducción y la regla (ITP_{proj}) el resultado se mantiene.

∎

Proposición 4.6.13 (Precisión monótona para $\circ^=$.)**.** *Si* $\varepsilon_1 \sqsubseteq \varepsilon_2$ *y* $\varepsilon_3 \sqsubseteq \varepsilon_4$ *entonces* $\varepsilon_1 \circ^= \varepsilon_3 \sqsubseteq \varepsilon_2 \circ^= \varepsilon_4$.

Demostración. La demostración es directa usando las definiciones de funciones de precisión. ∎

Proposición 4.6.14 (Gradual Meet preserva la Precisión de Tipos)**.** *Si* $U_{11} \sqsubseteq U_{12}$ *y* $U_{21} \sqsubseteq U_{22}$ *entonces* $U_{11} \sqcap U_{21} \sqsubseteq U_{12} \sqcap U_{22}$.

Demostración. La demostración se realiza por inducción sobre la derivación de tipos y *gradual meet*. El siguiente análisis es para el tipo registro (*record type*).
Usando la proposición de precisión de tipos para registros, Proposición 4.5.4 la relación $U_{11} \sqsubseteq U_{12}$ es tal que $[\ell_1 : U_{11}, \ldots, \ell_n : U_{1n}] \sqsubseteq [\ell_1 : U_{21}, \ldots, \ell_n : U_{2n}]$ con $U_{11} \sqsubseteq U_{21}, \ldots, U_{1n} \sqsubseteq U_{2n}$.
Sin pérdida de generalidad, considérense otros dos registros tales que $[\ell_1 : U_{31}, \ldots, \ell_n : U_{3n}] \sqsubseteq [\ell_1 : U_{41}, \ldots, \ell_n : U_{4n}]$.
Usando la definición de *gradual meet* (Definición 4.5.8), el *meet* de los siguientes registros es un punto fijo:
$[\ell_1 : U_{11}, \ldots, \ell_n : U_{1n}] \sqcap [\ell_1 : U_{31}, \ldots, \ell_n : U_{3n}] = [\ell_1 : (U_{11} \sqcap U_{31}), \ldots, \ell_n : (U_{1n} \sqcap U_{3n})]$ y
$[\ell_1 : U_{21}, \ldots, \ell_n : U_{2n}] \sqcap [\ell_1 : U_{41}, \ldots, \ell_n : U_{4n}] = [\ell_1 : (U_{41} \sqcap U_{41}), \ldots, \ell_n : (U_{4n} \sqcap U_{4n})]$. El resultado se mantiene siguiendo las hipótesis de inducción. ∎

Proposición 4.6.15 (Garantía Dinámica para $\to$)**.** *Supongamos* $\Omega \vdash t_1^{U_1} \sqsubseteq t_1^{U_2}$. *Si* $t_1^{U_1} \to t_2^{U_1}$ *entonces* $t_1^{U_2} \to t_2^{U_2}$ *donde*
$\Omega' \vdash t_2^{U_1} \sqsubseteq t_2^{U_2}$ *para alguna* $\Omega' \supseteq \Omega$.

Demostración. La demostración se realiza por inducción sobre la estructura de $\Omega \vdash t_1^{U_1} \sqsubseteq t_1^{U_2}$ y la reducción $\to$.

- **Caso** (ITP_{proj})
  Considérese $\Omega \vdash \varepsilon_1\big(t_1^{U_1} \bullet^{[\ell_{1i}:U_{1i}\ i \in 1..n]} \ell_j\big) \sqsubseteq \varepsilon_2\big(t_1^{U_2} \bullet^{[\ell_{2i}:U_{2i}\ i \in 1..n]} \ell_j\big)$ que por la regla (ITP_{proj}), $\Omega \vdash t_1^{U_1} \sqsubseteq t_1^{U_2}$ se mantiene para $\varepsilon_1 \sqsubseteq \varepsilon_2$. Ahora, para reducir los siguientes términos, los registros deben tener la forma de términos evidencia (*evidence terms*). Supóngase que la reducción de la primera proyección es $\varepsilon_1 t_1^{U_1} \bullet^{[\ell_{1i}:U_{1i}\ i \in 1..n]} \ell_j \to iproj(\varepsilon_1, \ell_j) u_j :: U_{1j}$

La reducción $\varepsilon_2 t_1^{U_2} \bullet^{[\ell_{2i}:U_{2i}\ ^{i\in 1..n}]} \ell_j \to iproj(\varepsilon_2, \ell_j)u_j :: U_{2j}$ es realizable.
Entonces la relación $iproj(\varepsilon_1, \ell_j)u_j :: U_{1j} \sqsubseteq iproj(\varepsilon_2, \ell_j)u_j :: U_{2j}$ se mantiene, si y sólo si $\Omega \vdash u_j^{U_{1j}} \sqsubseteq u_j^{U_{2j}}$ por la regla (ITP_{proj}) con $iproj(\varepsilon_1, \ell_j) \sqsubseteq iproj(\varepsilon_2, \ell_j)$, el cual se mantiene por la regla $(ITP_{::})$.

■

La siguiente Proposición 4.6.16 significa que:

"...no solo los programas más dinámicos continúan ejecutándose al menos tan bien como sus contrapartes más estáticas, sino también los programas más estáticos continúan ejecutándose al menos igual de mal: agregar más tipos estáticos no soluciona los errores de tipo anteriores" [42].

$$\begin{array}{ccc} t_1^{U_1} & \sqsubseteq & t_1^{U_2} \\ \downarrow & & \downarrow \\ t_2^{U_1} & \sqsubseteq & t_2^{U_2} \end{array}$$

En otras palabras, esta proposición implica que el comportamiento de dos programas durante el tiempo de ejecución es comparable bajo la noción de precisión. La proposición es, en efecto, una bisimulación [42], donde es posible observar las ejecuciones de los términos intrínsecos y mostrar el término precisión en cada paso de la evaluación:

Proposición 4.6.16 (Garantía dinámica)**.** *Supongamos* $t_1^{U_1} \sqsubseteq t_1^{U_2}$*. Si* $t_1^{U_1} \mapsto t_2^{U_1}$ *entonces* $t_1^{U_2} \mapsto t_2^{U_2}$ *donde* $t_2^{U_1} \sqsubseteq t_2^{U_2}$*.*

Demostración. La demostración es por inducción sobre la derivación de $t_1^{U_1} \sqsubseteq t_1^{U_2}$. La demostración está en [95]. ■

Cálculo de Conversión de Tipos

El Cálculo de Conversión de Tipos (en inglés *Cast Calculus o CC)*) se inserta durante el tiempo de ejecución para preservar apropiadamente la consistencia de tipos en un sistema tipificado gradual. La semántica durante tiempo de ejecución se define en términos de estas conversiones explícitas. Es importante recordar que la propiedad de consistencia expone que el valor de un término debe tener el tipo esperado [25].

La semántica de un lenguaje con tipificado gradual normalmente está dada por una traducción a un lenguaje intermedio con *casts* (conversiones de tipos), es decir, durante el tiempo de ejecución una comprobación de tipos controla los límites entre el código tipificado y no tipificado [3]. La comprobación de tipos no es una tarea sencilla. En particular, para los lenguajes tipificados dinámicamente y los lenguajes tipificados gradualmente, ya que se debe garantizar el rendimiento de los programas y la seguridad de tipos de los lenguajes [3, 75]. El enfoque de tipificado gradual de tipo unión utiliza una traducción de inserción de *cast* en un lenguaje intermedio utilizando operaciones de conversión de tipos explícitos. Es importante decir que el Cálculo de Conversiones de Tipos tiene un lenguaje objetivo intermedio obtenido del Cálculo Lambda de Tipificado Gradual [76]. Esto ayuda a hacer la transición de tipos entre dos términos cualesquiera, haciendo explícita esta conversión, donde a su vez, se debe de preservar la consistencia del lenguaje. En este punto, la consistencia de un lenguaje significa que en tiempo de ejecución un programa debe regresar un valor de algún tipo esperado, por lo que durante las operaciones de conversión de tipos en tiempo de ejecución el tipo esperado debe conservarse. Para realizar de manera adecuada un *cast* durante tiempo de ejecución, el Sistema de Tipos Gradual debe agregar una inserción correcta de *cast* [25, 79].

El uso de *cast* para integrar la tipificación estática y dinámica en un lenguaje debe asegurar la convergencia de dos tipos. Por un lado, la conversión de tipos generalmente se describe entre dos tipos: un tipo origen o fuente, y un tipo de destino. A esto se le conoce como **twosome**, porque sólo se utilizan dos factores en la operación de *cast*. Por otro lado, se dice que si la conversión de tipos realiza un *downcast*, el cual se hace desde un tipo origen a un tipo intermedio; y a su vez se realiza un *upcast* entre el tipo intermedio al tipo destino, a esto se le conoce como **threesome**, debido a que se necesitan tres tipos para realizar una conversión de tipos [84]. Un *threesome* tiene la forma $\langle T \overset{l}{\Longleftarrow} S \rangle$ donde T denota un tipo destino, S un tipo de origen, y l denota una etiqueta asociada a la conversión de tipos. En algunos casos, la etiqueta no es relevante en el *cast* y, por lo tanto, se omite.

Además, un *threesome* continúa la idea de contener una triada, en el sentido de que los elementos principales (tipo origen y tipo destino) se conservan bajo esta relación. El tipo intermedio o medio está representado en el

enfoque GUT por el cálculo de la cota máxima inferior, conocida en inglés como *meet*, es decir, la etiqueta l en el *cast* de los dos tipos es el ***meet* de los dos extremos** [42, 84]. Siguiendo el enfoque de [95], la notación utilizada para *threesome* $\langle T \overset{T \sqcap S}{\Longleftarrow} S \rangle$ tiene el mismo significado, donde un tipo intermedio está representado por el *meet* de ambos tipos (objetivo y origen).

La Figura 4.12 describe el sistema de tipos completo para el lenguaje intermedio $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$ extendido con registros mediante las reglas (IT_{rec}) y (IT_{proj}). Observemos que se realiza el *cast* tanto para los términos como para los valores en $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$, es decir, se realizan las conversiones de tipos de $\langle U \overset{U}{\Longleftarrow} U \rangle t$ y de $\langle U \overset{U}{\Longleftarrow} U \rangle u$ correspondientes. También es importante expresar que el término referido a la igualdad de tipos se utiliza en lugar de la consistencia de tipos debido a una consistencia asegurada por la inserción de los *casts* [95].

Una inserción de *cast* se aplica a un término t con algún tipo origen U_1, el cual puede convertirse en otro tipo destino U_2 utilizando un tipo intermedio U_3. Este último tipo U_3, que se utiliza en medio de esta conversión, es el *meet* o la mayor cota inferior (en términos de precisión) del tipo U_1 y U_2. Esta conversión se basa en el **Cálculo Threesome Original** sin utilizar el Cálculo de la Culpa (*Blame Calculus*) de [95].

$$\langle\langle U_2 \overset{U_3}{\Longleftarrow} U_1\rangle\rangle t = \begin{cases} t & \text{si } U_1 = U_2 = U_3 \\ \langle U_2 \overset{U_3}{\Longleftarrow} U_1\rangle t & \text{en otro caso} \end{cases}$$

$$U \in TYPE, \quad x \in VAR, \quad t \in TERM, \quad \Gamma \in VAR \rightharpoonup^{fin} TYPE$$

$$\begin{array}{rcll} U & ::= & Bool \mid Int \mid U \to U \mid [\ell_i : U_i{}^{i\in 1..n}] & \text{(tipos)} \\ u & ::= & n \mid b \mid (\lambda x : U.t) \mid [\ell_i = u_i{}^{i\in 1..n}] & \text{(valores simples)} \\ v & ::= & u \mid \langle U \overset{U}{\Longleftarrow} U\rangle u & \text{(valores)} \\ t & ::= & v \mid x \mid tt \mid t + t \mid \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ t \mid t :: T & \\ & \mid & \langle U \overset{U}{\Longleftarrow} U\rangle t \mid [\ell_i = t_i{}^{i\in 1..n}] \mid t.\ell_j & \text{(términos)} \end{array}$$

$$\frac{}{\Gamma \vdash_{\Leftarrow} n : Int}\ (IT_n) \qquad \frac{}{\Gamma \vdash_{\Leftarrow} b : Bool}\ (IT_b) \qquad \frac{x : U \in \Gamma}{\Gamma \vdash_{\Leftarrow} x : U}\ (IT_x)$$

$$\frac{\Gamma \vdash_{\Leftarrow} t_1 : Int \qquad \Gamma \vdash_{\Leftarrow} t_2 : Int}{\Gamma \vdash_{\Leftarrow} t_1 + t_2 : Int}\ (IT_+) \qquad \frac{\Gamma \vdash_{\Leftarrow} t_1 : Bool \qquad \Gamma \vdash_{\Leftarrow} t_2 : U_2 \qquad \Gamma \vdash_{\Leftarrow} t_3 : U_2}{\Gamma \vdash_{\Leftarrow} \mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3 : U_2}\ (IT_{if})$$

$$\frac{\Gamma \vdash_{\Leftarrow} t : U_1 \qquad U_1 \sim U_2 \qquad U_1 \sim U_3 \qquad U_3 \sim U_2}{\Gamma \vdash_{\Leftarrow} \langle U_2 \overset{U_3}{\Longleftarrow} U_1\rangle t : U_2}\ (IT_{\langle\rangle})$$

$$\frac{\Gamma, x : U_1 \vdash_{\Leftarrow} t : U_2}{\Gamma \vdash_{\Leftarrow} (\lambda x : U_1.t) : U_1 \to U_2}\ (IT_\lambda) \qquad \frac{\Gamma \vdash_{\Leftarrow} t_1 : U_1 \qquad \Gamma \vdash_{\Leftarrow} t_2 : \widetilde{dom}(U_1)}{\Gamma \vdash_{\Leftarrow} t_1 t_2 : \widetilde{cod}(U_1)}\ (IT_{app})$$

$$\frac{\Gamma \vdash_{\Leftarrow} t_i : U_i \quad \textit{para cada } i \in 1..n}{\Gamma \vdash_{\Leftarrow} [\ell_i = t_i{}^{i\in 1..n}] : [\ell_i : U_i{}^{i\in 1..n}]}\ (IT_{rec}) \qquad \frac{\Gamma \vdash_{\Leftarrow} t : U}{\Gamma \vdash_{\Leftarrow} t.\ell_j : \widetilde{proj}(U, \ell_j)}\ (IT_{proj})$$

Figura 4.12: Sintaxis y Sistema de Tipos del lenguaje intermedio $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$.

A continuación se exponen algunos ejemplos de derivaciones de tipos usando el sistema de tipos en $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$ definido en la Figura 4.12.

Primer ejemplo usando (IT_+, IT_n e IT_n)

$$\frac{\dfrac{}{\Gamma \vdash_{\Leftarrow} 2 : Int}\,(IT_n) \qquad \dfrac{}{\Gamma \vdash_{\Leftarrow} 3 : Int}\,(IT_n)}{\Gamma \vdash_{\Leftarrow} 2 + 3 : Int}\,(IT_+)$$

Segundo ejemplo usando (IT_{if}, IT_b, IT_n, e IT_n)

$$\frac{\dfrac{}{\Gamma \vdash_{\Leftarrow} true : Bool}\,(IT_b) \quad \dfrac{}{\Gamma \vdash_{\Leftarrow} 1 : Int}\,(IT_n) \quad \dfrac{}{\Gamma \vdash_{\Leftarrow} 2 : Int}\,(IT_n)}{\Gamma \vdash_{\Leftarrow} if\ true\ then\ 1\ else\ 2 : Int}\,(IT_{if})$$

Tercer ejemplo usando (IT_{app}, IT_λ, IT_b, IT_{proj}, IT_{rec}, IT_n e IT_b)

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : Int \oplus Bool \vdash_{\Leftarrow} 1 : Int}\,(IT_n)\ \dfrac{}{\Gamma, x : Int \oplus Bool \vdash_{\Leftarrow} true : Bool}\,(IT_b)}{\Gamma, x : Int \oplus Bool \vdash_{\Leftarrow} [\ell_1 = 1, \ell_2 = true] : [\ell_1 : Int, \ell_2 : Bool]}\,(IT_{rec})}{\Gamma, x : Int \oplus Bool \vdash_{\Leftarrow} [\ell_1 = 1, \ell_2 = true].\ell_2 : Bool}\,(IT_{proj})}{\Gamma \vdash_{\Leftarrow} (\lambda x : Int \oplus Bool.\, ([\ell_1 = 1, \ell_2 = true].\ell_2)) : Int \oplus Bool \to Bool}\,(IT_\lambda) \quad \dfrac{}{\Gamma \vdash_{\Leftarrow} true : (Int \oplus Bool)}\,(IT_b)}{\Gamma \vdash_{\Leftarrow} ((\lambda x : Int \oplus Bool.\, ([\ell_1 = 1, \ell_2 = true].\ell_2))\ true) : Bool}\,(IT_{app})$$

Cuarto ejemplo usando (IT_{app}, IT_λ, IT_x, IT_{proj}, IT_{rec}, IT_n e IT_b)

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x : Int \oplus ? \to Bool \vdash_{\Leftarrow} 1 : Int}\,(IT_n) \quad \dfrac{}{\Gamma, x : Int \oplus ? \to Bool \vdash_{\Leftarrow} true : Bool}\,(IT_b)}{\Gamma, x : Int \oplus ? \to Bool \vdash_{\Leftarrow} [\ell_1 = 1, \ell_2 = true] : [\ell_1 : Int, \ell_2 : Bool]}\,(IT_{rec})}{\Gamma, x : Int \oplus ? \to Bool \vdash_{\Leftarrow} [\ell_1 = 1, \ell_2 = true].\ell_2 : Bool}\,(IT_{proj})}{\Gamma \vdash_{\Leftarrow} (\lambda x : Int \oplus ?.\, ([\ell_1 = 1, \ell_2 = true].\ell_2)) : Int \oplus ? \to Bool}\,(IT_\lambda) \quad \dfrac{y : (Int \oplus ?) \in \Gamma}{\Gamma \vdash_{\Leftarrow} y : (Int \oplus ?)}\,(IT_x)}{\Gamma \vdash_{\Leftarrow} \big((\lambda x : Int \oplus ?.\, ([\ell_1 = 1, \ell_2 = true].\ell_2))\ y\big) : Bool}\,(IT_{app})$$

Figura 4.13: Algunos ejemplos de derivaciones usando el lenguaje intermedio de $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$.

La Figura 4.14 describe la semántica dinámica de $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$ donde los valores simples denotados por u, no tienen anotaciones de tipos. Los valores v son cualquier tipo de valor simple u o un valor simple con una operación de conversión de tipos (*cast*) $\langle U \overset{U}{\Longleftarrow} U \rangle u$. La semántica operativa de los términos graduales intrínsecos se define mediante *frames* para establecer el orden de reducción de un término. La noción de reducción expresa, en primer lugar los casos de valores simples mediante la metafunción *rval* para acceder al valor subyacente, lo que implica eliminar las conversiones de tipos redundantes. La aplicación y la proyección de registros tienen las reducciones habituales. La reducción de valores con *cast* inserta una reducción (de conversión de tipos) utilizando las funciones *icod* y *idom*. Por último, un valor con *cast* compuesto se reduce mediante una combinación de ellos: dos *threesomes* que coinciden en sus tipos (origen y destino) si combinan sus tipos intermedios usando el *meet*. El resultado de ambas conversiones de tipos, se fusiona ahora para obtener un nuevo *cast*, el tipo intermedio de los otros tipos medios.

A continuación se muestra la semántica de $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$.

$$
\begin{array}{lcll}
u & ::= & n \mid true \mid false \mid \lambda x.t \mid [\ell_i = {u_i}^{i\in 1..n}] & (valores\ simples)\\
v & ::= & u \mid \langle U \overset{U}{\Longleftarrow} U\rangle u & (valores)\\
f & ::= & \Box + t \mid v + \Box \mid \Box t \mid v\Box \mid \langle U \overset{U}{\Longleftarrow} U\rangle \Box \mid if\ \Box\ then\ t\ else\ t & \\
 & \mid & [\ell_i = {v_i}^{i\in 1..j+1}, \ell_j = \Box, \ldots, \ell_k = {t_k}^{k\in j+1..n}] \mid \Box.\ell_j & (marcos)
\end{array}
$$

Noción de Reducción: $\boxed{t \to t}$

$$
\begin{array}{rcl}
v_1 + v_2 & \to & n_3 \qquad \text{donde } n_3 = rval(v_1)[\![+]\!]rval(v_2)\\
if\ v\ then\ t_1\ else\ t_2 & \to & \begin{cases} t_1 & \text{si } rval(v) = true \\ t_2 & \text{si } rval(v) = false \end{cases}\\
(\lambda x.t)v & \to & t[v/x]\\
[\ell_i = {v_i}^{i\in 1..n}].\ell_j & \to & v_j\\
\langle U_{21} \to U_{22} \overset{U_3}{\Longleftarrow} U_{11} \to U_{12}\rangle u\, v & \to & \langle\langle U_{22} \overset{\widetilde{cod}(U_3)}{\Longleftarrow} U_{12}\rangle\rangle\Big(u\langle\langle U_{11} \overset{\widetilde{dom}(U_3)}{\Longleftarrow} U_{21}\rangle\rangle v\Big)\\
\langle U_1 \overset{U_3}{\Longleftarrow} U_2\rangle v.\ell_j & \to & \begin{cases} \langle\langle U_{1j} \overset{\widetilde{proj}(U_3,\ell_j)}{\Longleftarrow} U_{2j}\rangle\rangle(v.\ell_j) \\ \textbf{error si } \widetilde{proj}(U_3,\ell_j) \text{ es indefinido} \end{cases}\\
\langle U_3 \overset{U_{32}}{\Longleftarrow} U_2\rangle\langle U_2 \overset{U_{21}}{\Longleftarrow} U_1\rangle v & \to & \begin{cases} \langle\langle U_3 \overset{U_{32}\sqcap U_{21}}{\Longleftarrow} U_1\rangle\rangle v \\ \textbf{error si } U_{32} \sqcap U_{21} \text{ es indefinido} \end{cases}
\end{array}
$$

Reglas de Reducción: $\boxed{t \mapsto t}$

$$
\frac{t_1 \to t_2}{t_1 \mapsto t_2} \qquad \frac{t_1 \mapsto t_2}{f[t_1] \mapsto f[t_2]} \qquad \frac{}{f[\textbf{error}] \mapsto \textbf{error}}
$$

Figura 4.14: Semántica Dinámica de $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$.

La Figura 4.15 muestra las reglas de traducción para la inserción de conversión de tipos, que se define por los juicios de tipo $\Gamma \vdash \widetilde{t} \Rightarrow t : U$ para traducir términos entre $\mathsf{GTFL}^{\oplus}_{rec}$ y $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$. Obsérvese que el símbolo $\Rightarrow$ en las traducciones expresa la transformación necesaria para preservar la consistencia del lenguaje de términos intrínsecos $\widetilde{t}$ a términos intermedios con *cast*. La nomenclatura para las operaciones de conversión de tipos utilizada aquí se expresa por medio de un *twosome* $\langle U_2 \Longleftarrow U_1\rangle t$ el cual es usado en lugar de $\langle\langle U_2 \overset{U_1\sqcap U_2}{\Longleftarrow} U_1\rangle\rangle t$, sin embargo, cabe resaltar que la semántica de *threesome* se matiene. La idea principal es aplicar la conversión de tipos (*cast*) para cada restricción de consistencia de tipos en las premisas. El *cast* resume los tipos de unión gradual para ser utilizados durante el tiempo de ejecución, es decir, una conversión de tipos produce una inyección implícita al *gradual meet* (dada en la Definición 4.5.8).

La extensión con registros incluye las reglas C_{rec} y C_{proj}. La primera supone que para cada $i \in 1..n$, los términos intrínsecos $\widetilde{t_i}$ ya están traducidos a los correspondientes términos intermedios $t_i^{'}$ y son usados para construir registros en los lenguajes correspondientes con el tipo $[\ell_i : {U_i}^{i\in 1..n}]$. Este último, C_{proj}, supone que $\widetilde{t}$ se traduce a $t^{'}$ asociado al tipo U_1 el cual es consistente con el tipo registro (*record*), por lo que la proyección de una etiqueta ℓ_j (para alguna $j \in 1..n$) sobre ambos registros resulta en un término de tipo U_j. La función *proj* se utiliza aquí para obtener el tipo correspondiente en U_1.

$$\frac{}{\Gamma \vdash n \Rightarrow n : Int}\ (C_n) \qquad \frac{}{\Gamma \vdash b \Rightarrow b : Bool}\ (C_b) \qquad \frac{x : U \in \Gamma}{\Gamma \vdash x \Rightarrow x : U}\ (C_x)$$

$$\frac{\Gamma \vdash \widetilde{t_1} \Rightarrow t_1^{'} : U_1 \qquad \Gamma \vdash \widetilde{t_2} \Rightarrow t_2^{'} : U_2 \qquad U_1 \sim Int \qquad U_2 \sim Int}{\Gamma \vdash \widetilde{t_1} + \widetilde{t_2} \Rightarrow \langle Int \Longleftarrow U_1 \rangle t_1^{'} + \langle Int \Longleftarrow U_2 \rangle t_2^{'} : Int}\ (C_+)$$

$$\frac{\Gamma \vdash \widetilde{t_1} \Rightarrow t_1^{'} : U_1 \qquad U_1 \sim Bool \qquad \Gamma \vdash \widetilde{t_2} \Rightarrow t_2^{'} : U_2 \qquad \Gamma \vdash \widetilde{t_3} \Rightarrow t_3^{'} : U_3}{\Gamma \vdash \mathit{if}\ \widetilde{t_1}\ \mathit{then}\ \widetilde{t_2}\ \mathit{else}\ \widetilde{t_3} \Rightarrow \mathit{if}\ \langle Bool \Longleftarrow U_1 \rangle t_1^{'}\ \mathit{then}\ \langle U_2 \sqcap U_3 \Longleftarrow U_2 \rangle t_2^{'}\ \mathit{else}\ \langle U_2 \sqcap U_3 \Longleftarrow U_3 \rangle t_3^{'} : U_2 \sqcap U_3}\ (C_{if})$$

$$\frac{\Gamma \vdash \widetilde{t} \Rightarrow t^{'} : U \qquad U \sim U_1}{\Gamma \vdash (\widetilde{t} :: U_1) \Rightarrow \langle U_1 \Longleftarrow U \rangle t^{'} : U_1}\ (C_{::}) \qquad \frac{\Gamma, x : U_1 \vdash \widetilde{t} \Rightarrow t^{'} : U_2}{\Gamma \vdash (\lambda x : U_1 . \widetilde{t}) \Rightarrow (\lambda x : U_1 . t^{'}) : U_1 \to U_2}\ (C_\lambda)$$

$$\frac{\Gamma \vdash \widetilde{t_1} \Rightarrow t_1^{'} : U_1 \qquad \Gamma \vdash \widetilde{t_2} \Rightarrow t_2^{'} : U_2 \qquad U_2 \sim \widetilde{dom}(U_1)}{\Gamma \vdash \widetilde{t_1}\widetilde{t_2} \Rightarrow \langle \widetilde{dom}(U_1) \to \widetilde{cod}(U_1) \Longleftarrow U_1 \rangle t_1^{'} \langle \widetilde{dom}(U_1) \Longleftarrow U_2 \rangle t_2^{'} : \widetilde{cod}(U_1)}\ (C_{app})$$

$$\frac{\Gamma \vdash \widetilde{t_i} \Rightarrow t_i^{'} : U_i \qquad para\ cada\ i \in 1..n}{\Gamma \vdash [\ell_i = \widetilde{t_i}^{\,i \in 1..n}] \Rightarrow [\ell_i = {t'}_i^{\,i \in 1..n}] : [\ell_i : {U_i}^{i \in 1..n}]}\ (C_{rec})$$

$$\frac{\Gamma \vdash \widetilde{t} \Rightarrow t^{'} : U \qquad U \sim [\ell_i : {U_i}^{i \in 1..n}]}{\Gamma \vdash \widetilde{t}.\ell_j \Rightarrow \langle [\ell_i : {U_i}^{i \in 1..n}] \Longleftarrow U \rangle t'.\ell_j : \widetilde{proj}(U, \ell_j)}\ (C_{proj})$$

Figura 4.15: Reglas de Traducción para la Inserción de *Cast*.

A continuación, se exponen algunos ejemplos de derivaciones de tipos usando las conversiones de tipos en el sistema $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$ definido en la Figura 4.15.

Primer ejemplo usando (C_{app}, C_λ, C_b, C_{proj}, C_{rec}, C_n y C_b)

$$\dfrac{\dfrac{}{\Gamma, x : Int \oplus Bool \vdash 1 : Int}\,(C_n)\quad \dfrac{}{\Gamma, x : Int \oplus Bool \vdash true : Bool}\,(C_b)}{\Gamma, x : Int \oplus Bool \vdash [\ell_1 = 1, \ell_2 = true] \Rightarrow [\ell_1 = 1, \ell_2 = true] : [\ell_1 : Int, \ell_2 : Bool]}\,(C_{rec})$$

$$\dfrac{\dfrac{[\ell_1 : Int, \ell_2 : Bool] \sim [\ell_1 : Int, \ell_2 : Bool]}{\Gamma, x : Int \oplus Bool \vdash ([\ell_1 = 1, \ell_2 = true].\ell_2) \Rightarrow <[\ell_1 : Int, \ell_2 : Bool] \Leftarrow [\ell_1 : Int, \ell_2 : Bool]>([\ell_1 = 1, \ell_2 = true].\ell_2) \Rightarrow Bool}\,(C_{proj})}{\Gamma \vdash \big(\lambda x : Int \oplus Bool.\, ([\ell_1 = 1, \ell_2 = true].\ell_2)\big) \Rightarrow \big(\lambda x : Int \oplus Bool.\, ([\ell_1 = 1, \ell_2 = true].\ell_2)\big) : Int \oplus Bool \to Bool}\,(C_\lambda)$$

$$\dfrac{Bool \sim Int \oplus Bool \qquad \dfrac{}{\Gamma \vdash true \Rightarrow true : Bool}\,(C_b)}{\begin{array}{c}\Gamma \vdash ((\lambda x : Int.\, ([\ell_1 = 1, \ell_2 = true].\ell_2)) \quad true) : Bool \Rightarrow \\ <Int \oplus Bool \to Bool \Leftarrow Int \oplus Bool>\big((\lambda x : Int \oplus Bool.([\ell_1 = 1, \ell_2 = true]).\ell_2) <Int \oplus Bool \Leftarrow Bool> \quad true\big) : Bool\end{array}}\,(C_{app})$$

Segundo ejemplo usando (C_{app}, C_λ, C_x, C_{proj}, C_{rec}, C_n y C_b)

$$\dfrac{\dfrac{}{\Gamma, x : Int \oplus ? \vdash 1 : Int}\,(C_n)\quad \dfrac{}{\Gamma, x : Int \oplus ? \vdash true : Bool}\,(C_b)}{\Gamma, x : Int \oplus ? \vdash [\ell_1 = 1, \ell_2 = true] \Rightarrow [\ell_1 = 1, \ell_2 = true] : [\ell_1 : Int, \ell_2 : Bool]}\,(C_{rec})$$

$$\dfrac{\dfrac{[\ell_1 : Int, \ell_2 : Bool] \sim [\ell_1 : Int, \ell_2 : Bool]}{\Gamma, x : Int \oplus ? \vdash ([\ell_1 = 1, \ell_2 = true].\ell_2) \Rightarrow <[\ell_1 : Int, \ell_2 : Bool] \Leftarrow [\ell_1 : Int, \ell_2 : Bool]>([\ell_1 = 1, \ell_2 = true].\ell_2) \Rightarrow Bool}\,(C_{proj})}{\Gamma \vdash \big(\lambda x : Int \oplus ?.\, ([\ell_1 = 1, \ell_2 = true].\ell_2)\big) \Rightarrow \big(\lambda x : Int \oplus ?.\, ([\ell_1 = 1, \ell_2 = true].\ell_2)\big) : Int \oplus ? \to Bool}\,(C_\lambda)$$

$$\dfrac{? \sim Int \oplus ? \qquad \dfrac{y : ? \in \Gamma}{\Gamma \vdash y \Rightarrow y : ?}\,(C_x)}{\begin{array}{c}\Gamma \vdash ((\lambda x : Int \oplus ?.\, ([\ell_1 = 1, \ell_2 = true].\ell_2)) \quad y) : Bool \Rightarrow \\ \big(<Int \oplus ? \to Bool \Leftarrow Int \oplus ?>\big(\lambda x : Int \oplus ?.([\ell_1 = 1, \ell_2 = true]).\ell_2\big) <(Int \oplus ?) \Leftarrow ?> \quad y\big) : Bool\end{array}}\,(C_{app})$$

Figura 4.16: Algunos ejemplos de derivaciones de tipos usando las inserciones de conversión de tipos $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$.

Propiedades de la traducción semántica

En esta sección se presentan las propiedades y las definiciones necesarias para demostrar la corrección de la semántica de traducción, basada en las siguientes relaciones lógicas entre $\mathsf{GTFL}^{\oplus}_{rec}$ y el lenguaje intermedio $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$.

Las relaciones lógicas describen la relación estática y dinámica entre dos términos: $\check{t}$ en $\mathsf{GTFL}^{\oplus}_{rec}$ (o $\widetilde{t}$ [20]) y t en $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$, para denotar que ambos términos dan cálculos relacionados en k pasos respetando el tipo U. Estas relaciones se definen para diferentes categorías de la siguiente manera. Cuando dos elementos están relacionados $(e_1, e_2) \in \mathcal{R}_k[\![U]\!]$ significa que ambos elementos pertenecen a la misma categoría sintáctica R en k pasos de reducción y el tipo correspondiente está dado por U.

$$(b_1, b_2) \in \mathcal{U}_k[\![Bool]\!] \iff b_1 \in TERM_{Bool} \wedge b_2 : Bool \wedge b_1 = b_2$$

$$(n_1, n_2) \in \mathcal{U}_k[\![Int]\!] \iff n_1 \in TERM_{Int} \wedge n_2 : Int \wedge n_1 = n_2$$

$$(\check{u_1}, u_2) \in \mathcal{U}_k[\![U_1 \to U_2]\!] \iff \begin{cases} \check{u_1} \in TERM_{U_1 \to U_2} \wedge u_2 : U_1 \to U_2 \wedge \\ \big(\forall\, U' = U_1'' \to U_2'',\ \varepsilon_1 \vdash U_1 \to U_2 \sim U_1'' \to U_2''\ \text{ y } \varepsilon_2 \vdash U_1' \sim U_1'', \\ \text{tenemos: } \forall j \le k,\ (\check{v_1}, v_2) \in \mathcal{V}_j[\![U_1']\!], \\ (\varepsilon_1 \check{u_1} @^{U'}\ \varepsilon_2 \check{v_1},\ \langle U' \xleftarrow{\varepsilon_1} U_1 \to U_2\rangle u_2 \langle U_1'' \xleftarrow{\varepsilon_2} U_1'\rangle v_2) \in \mathcal{T}_j[\![U_2'']\!]\big) \end{cases}$$

$$(\check{u_1}, u_2) \in \mathcal{U}_k[\![[\ell_i : U_i{}^{\,i\in 1..n}]]\!] \iff \begin{cases} \check{u_1} \in TERM_{[\ell_i : U_i\ ^{i\in 1..n}]} \wedge u_2 : [\ell_i : U_i{}^{\,i\in 1..n}] \\ \wedge \big(\forall\, U' = [\ell_i : U''_i{}^{\,i\in 1..n}] \wedge \varepsilon \vdash [\ell_i : U_i{}^{\,i\in 1..n}] \sim [\ell_i : U''_i{}^{\,i\in 1..n}] \\ \text{tenemos: } \forall k' \le k, \\ (\varepsilon \check{u_1} \bullet^{U'} \ell_j, \langle U' \xleftarrow{\varepsilon} [\ell_i : U_i{}^{\,i\in 1..n}]\rangle u_2.\ell_j) \in \mathcal{T}_{k'}[\![U_j'']\!] \text{ para toda } j \in 1..n\big) \end{cases}$$

$$(\varepsilon \check{u_1} :: U, u_2) \in \mathcal{V}_k[\![U]\!] \iff \varepsilon \check{u_1} :: U \in TERM_U \wedge \varepsilon = U \wedge (\check{u_1}, u_2) \in \mathcal{U}_k[\![U]\!]$$

$$(\varepsilon \check{u_1} :: U, \langle U \xleftarrow{\varepsilon} U'\rangle u_2) \in \mathcal{V}_k[\![U]\!] \iff \varepsilon \check{u_1} :: U \in TERM_U \wedge (\check{u_1}, u_2) \in \mathcal{U}_{k+1}[\![U']\!]$$

$$(\check{u_1}, u_2) \in \mathcal{V}_k[\![U]\!] \iff (\check{u_1}, u_2) \in \mathcal{U}_k[\![U]\!]$$

$$(\check{t_1}, t_2) \in \mathcal{T}_k[\![U]\!] \iff \begin{cases} \check{t_1} \in TERM_U \wedge \vdash t_2 : U \wedge \big(\forall j < k \\ \big(\check{t_1} \mapsto^j \check{v_1} \Rightarrow (t_2 \mapsto^\star v_2 \wedge (\check{v_1}, v_2) \in \mathcal{V}_{k-j}[\![U]\!])\big) \wedge \\ \big(t_2 \mapsto^j v_2 \Rightarrow (\check{t_1} \mapsto^\star \check{v_1} \wedge (\check{v_1}, v_2) \in \mathcal{V}_{k-j}[\![U]\!])\big) \wedge \\ (\check{t_1} \mapsto^j \mathbf{error} \Rightarrow t_2 \mapsto^\star \mathbf{error}) \wedge (\check{t_2} \mapsto^j \mathbf{error} \Rightarrow t_1 \mapsto^\star \mathbf{error})\big) \end{cases}$$

Figura 4.17: Relaciones lógicas entre términos intrínsecos y términos de *Cast Calculus*.

Estas relaciones lógicas se presentan en la Figura 4.17 y se definen por recursión mutua entre valores simples, valores y sus evaluaciones (aquí, para simplificar, la notación utilizada para los términos intrínsecos será $\check{t}$ en lugar de t^U o $\widetilde{t}$). Por ejemplo, para la categoría sintáctica de valores simples, el par $(\check{u_1}, u_2) \in \mathcal{U}_k[\![U]\!]$ expresa que un par de valores simples $(\check{u_1}, u_2)$ está relacionado bajo el tipo U en k pasos, si ambos valores tienen el mismo tipo U. La relación se mantiene para un par de valores $(\check{v_1}, v_2)$, es decir, entre un valor intrínseco $\check{v_1}$ y un valor $v \in \mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$. Esta relación expresa que ambos términos están relacionados en k pasos si tienen el mismo tipo U. Ahora, para un valor adscrito intrínseco se dice que está relacionado con un valor simple u_2 si su tipo es exactamente U, o bien se realiza un *cast* del tipo correspondiente para preservar la consistencia del tipo. Y se dice que dos términos están relacionados en sus cálculos o cómputos $(\check{t_1}, t_2)$ en k pasos, a través de la relación lógica $\mathcal{T}_k[\![U]\!]$ donde ambos términos se reducen a valores relacionados de tipo U, o ambos se reducen a un error.

[20] Para fines de este trabajo, se usará la notación de $\check{t}$ la cual resulta más limpia y compacta, sobre todo en las demostraciones que se desarrollan en este apartado.

La siguiente definición establece la equivalencia semántica entre términos intrínsecos abiertos y términos intermedios abiertos relacionándolos en $\mathcal{T}_k[\![U]\!]$ mediante las sustituciones adecuadas.

Definición 4.8.1 (Equivalencia semántica). *Sea* $\check{t} \in TERM_U$, $\Gamma = FV\left(\check{t}\right)$ *y un término* t *en el lenguaje intermedio tal que* $\Gamma \vdash_{\Leftarrow} t : U$. *Los términos* $\check{t}$ *y* t *son semánticamente equivalentes, y los denotamos por* $\check{t} \approx t : U$, *si y sólo si para cualquier* $k \geq 0$ *de la sustitución* $(\sigma_1, \sigma_2) \in \mathcal{G}_k[\![\Gamma]\!]$ *se obtiene* $\left(\sigma_1(\check{t}), \sigma_2(t)\right) \in \mathcal{T}_k[\![U]\!]$.

Para demostrar la equivalencia semántica de los lenguajes $\mathsf{GTFL}^{\oplus}_{rec}$ y $\mathsf{GTFL}^{\oplus}_{rec,\Leftarrow}$, se proporciona la noción de sustitución y reducción de las relaciones lógicas dadas anteriormente. Las siguientes Definiciones 4.8.2 y 4.8.3 junto con los lemas 4.8.1, 4.8.2, 4.8.3 son necesarios para demostrar que la semántica de traducción es correcta. A continuación se incluyen los casos para los registros. Las demostraciones completas se pueden consultar en [95] respecto a algunas correcciones realizadas en el presente trabajo de tesis.

Definición 4.8.2 (Sustitución de Tipos en Ambientes (*Type Environment Substitution*)). *Sea* σ *una sustitución y* Γ *un ambiente. Decimos que la sustitución* σ *se satisface en el ambiente* Γ, *y lo denotamos como* $\sigma \vdash \Gamma$, *si y sólo si* $dom\,(\sigma) = dom\,(\Gamma)$.

La siguiente definición está relacionada con la noción de sustitución donde la notación $(\sigma_1, \sigma_2) \in \mathcal{G}_k[\![U]\!]$ describe dos sustituciones σ_1 y σ_2 que están relacionadas para k pasos en el tipo de ambiente Γ, si éstas mapean cada variable en Γ con algún valor relacionado [95].

Definición 4.8.3 (Sustituciones relacionadas). *Sea* σ_1 *una sustitución de variables intrínsecas a valores intrínsecos, y sea* σ_2 *una sustitución de variables a valores en el lenguaje intermedio. Entonces, definimos las sustituciones relacionadas como sigue:*

$$(\sigma_1, \sigma_2) \in \mathcal{G}_k[\![\Gamma]\!] \iff \sigma_i \vDash \Gamma \ \wedge \ \forall x \in \Gamma, \left(\sigma_1\left(x^{\Gamma(x)}\right), \sigma_2(x)\right) \in \mathcal{V}_k[\![\Gamma(x)]\!]$$

Lema 4.8.1 (La reducción preserva las relaciones). *Consideremos* $\Gamma \vdash \widetilde{t} : U$, $t^U \in TERM_U$ *y* $\Gamma \vdash \widetilde{t} \Rightarrow t : U$. *También, consideremos* $k, j > 0$, *si* $t^U \mapsto^j t'^U$ *y* $t \mapsto^j t'$, *entonces* $\left(t^U, t\right) \in \mathcal{T}_k[\![U]\!]$ *si y sólo si* $\left(t'^U, t'\right) \in \mathcal{T}_{k+j}[\![U]\!]$.

Demostración. Considérense las premisas $\Gamma \vdash \widetilde{t} : U$, $\Gamma \vdash \widetilde{t} \Rightarrow t : U$, $k, j > 0$ y las reducciones $t^U \mapsto^j t'^U$ y $t \mapsto^j t'$. Por la Proposición 4.6.5 se sabe que $t^U \mapsto^j t'$ con $t' \in CONFIG_U \cup \{\mathbf{error}\}$ y siguiendo reglas de la Figura 4.14, la reducción $t \mapsto^j t'$ se produce un valor o un error.

- Supongamos que $\left(t^U, t\right) \in \mathcal{T}_k[\![U]\!]$ entonces se quiere demostrar que $\left(t'^U, t'\right) \in \mathcal{T}_{k+j}[\![U]\!]$.
  Por definición de relación lógica se sabe que $\forall j' < k$, si $t^U \mapsto^{j'} {v_1}^U$ entonces $t \mapsto^\star v_2$, con $(v_1, v_2) \in \mathcal{V}_{k+j}[\![U]\!]$ y si $t \mapsto^j v_2$ entonces $t^U \mapsto^\star {v_1}^U$, con $(v_1, v_2) \in \mathcal{V}_{k+j}[\![U]\!]$ y si $t^U \mapsto^{j'} \{error\}$ entonces $t \mapsto^\star \{error\}$ y si $t \mapsto^{j'} \{error\}$ entonces $t \mapsto^\star \{error\}$. Por lo tanto, la relación se mantiene en $k - j$ pasos. Esto significa que (a) si t'^U es un término o un valor (dependiendo del caso), y t' es un valor, entonces la relación lógica se preserva. En otro caso (b) si t' es un término o un valor (dependiendo del caso), y ${t'}^U$ es un valor, y por lo tanto la relación lógica se preserva. Y (c) si t'^U o t' son errores ($error$), la relación lógica se preserva.

- Supongamos que $\left({t'}^U, t'\right) \in \mathcal{T}_{k-j}[\![U]\!]$, sigue la definición de la relación lógica, entonces tanto los términos son reducidos a valores de tipo U o producirán un error en $k + j$ pasos, donde para cualquier $j' < k - j$ entonces la relación lógica se preserva.
  Ahora para demostrar que $\left(t^U, t\right) \in \mathcal{T}_k[\![U]\!]$, se sabe que $t^U \mapsto^j {t'}^U$ y $t \mapsto^j t'$. La relación se preserva para j' incluyendo las reducciones en el paso j-ésimo, por lo que, para cualquier número de pasos menor a k se preserva.

■

Lema 4.8.2 (Términos relacionados). *Si* $\left(\widetilde{t_1}, t_2\right) \in \mathcal{T}_k[\![U]\!]$ *entonces si* $\varepsilon \vdash U \sim U'$, *entonces* $(\varepsilon\widetilde{t_1} :: U', \langle U' \overset{\varepsilon}{\Longleftarrow} U\rangle t_2) \in \mathcal{T}_{k+1}[\![U']\!]$.

Demostración. Considérese la relación lógica para términos $\left(\widetilde{t_1}, t_2\right) \in \mathcal{T}_k[\![U]\!]$ en la Figura 4.17. Los cómputos relacionados garantizan que en j pasos ($j < k$) ambos términos se reducen a valores en $\mathcal{V}_{k-j}[\![U]\!]$ o bien se envía un error.
En el primer caso, considérese $\varepsilon \vdash U \sim U'$ y entonces, para probar que $(\varepsilon\widetilde{t_1} :: U', \langle U' \overset{\varepsilon}{\Longleftarrow} U\rangle t_2) \in \mathcal{T}_{k+1}[\![U']\!]$ siguiendo la definición de valores simples donde $(\check{v_1}, v_2) \in \mathcal{U}_k[\![U]\!]$ y por lo tanto, los términos $\varepsilon\widetilde{t_1} :: U'$ y $\langle U' \overset{\varepsilon}{\Longleftarrow} U\rangle t_2$ están relacionados usando la relación lógica correspondientes para valores $(\varepsilon\check{u_1} :: U, \langle U' \overset{\varepsilon}{\Longleftarrow} U\rangle u_2) \in \mathcal{V}_{k+1}[\![U]\!]$.
En el segundo caso, la conclusión se mantiene dado que ambos términos se reducen a un error. ∎

Lema 4.8.3 (Cómputos relacionados)**.** *Consideremos $k > 0$. Si $\left(\widetilde{t_1}, t_2\right) \in \mathcal{T}_k[\![U]\!]$ entonces $\left(\widetilde{t_1}, t_2\right) \in \mathcal{T}_{k-1}[\![U]\!]$*

Demostración. La demostración es directa dada la definición de $(\check{t_1}, t_2) \in \mathcal{T}_k[\![U]\!]$. ∎

La siguiente proposición expone que si un término t está bien tipificado (desde el lenguaje gradual fuente), su término intrínseco correspondiente $\check{t}$ es semánticamente equivalente a un término intermedio t' obtenido después de la translación o traducción de la inserción de *cast* [95]. Esto significa que si el compilador puede realizar la verificación dinámica en un lenguaje con la garantía de tipificado gradual, esto es, que en tiempo de ejecución éste no se detenga, o dicho de otro modo, no pueda seguir evaluándose, entonces podemos decir que éste es un lenguaje con tipificado gradual **consistente** (del inglés *sound*) [19].

Proposición 4.8.1 (Equivalencia de referencia y semántica de traducción)**.** *Si $\Gamma \vdash \widetilde{t} : U$ está representado como un término intrínseco $\check{t} \in TERM_U$, y $\Gamma \vdash \widetilde{t} \Rightarrow t' : U$, entonces $\widetilde{t} \approx t' : U$*

Demostración. La demostración se realiza por inducción sobre la derivación de tipos de $\widetilde{t}$ (véase la Figura 4.6). Considérese $\Gamma \vdash \widetilde{t} : U$, $t^U \in TERM_U$, y $\Gamma \vdash \widetilde{t} \Rightarrow t : U$.
Entonces, dada la siguiente Definición 4.8.1, la demostración es: si $\forall k \geq 0$ y $(\sigma_1, \sigma_2) \in \mathcal{G}_k[\![\Gamma]\!]$ nos da $\left(\sigma_1\left(t^U\right), \sigma_2\left(t\right)\right) \in \mathcal{T}_k[\![U]\!]$.
Los siguientes son los casos para registros:

- **Caso** (U_{rec})
  Considérese $\widetilde{t} = [\ell_i = \widetilde{t_i}^{\,i\in 1..n}]$ representado como $\check{t} \in TERM_{[\ell_i : {U_i}^{\,i\in 1..n}]}$. Esto significa que para la siguiente Regla (U_{rec}), la hipótesis $\Gamma \vdash \widetilde{t_i} : U_i$ se mantiene para cada $\widetilde{t_i}$ y, a su vez por la Regla (IU_{rec}), $t_i^{U_i} \in TERM_{U_i}$ se mantiene.
  Para la siguiente regla (C_{rec}), el registro compilado es t' en $\Gamma \vdash \widetilde{t} \Rightarrow t' : [\ell_i : {U_i}^{\,i\in 1..n}]$ y cuyas hipótesis son $\Gamma \vdash \widetilde{t_i} \Rightarrow t_i' : U_i$ para cada $i \in 1..n$.
  Supóngase $k \geq 0$ y $(\sigma_1, \sigma_2) \in \mathcal{G}_k[\![\Gamma]\!]$, entonces, se demuestra que $\left(\sigma_1\left(\widetilde{t}\right), \sigma_2\left(t'\right)\right) \in \mathcal{T}_k[\![[\ell_i : {U_i}^{\,i\in 1..n}]]\!]$.

  Usando las hipótesis anteriores, y la Hipótesis de Inducción se mantienen para cada $\widetilde{t_i}$ y para toda $k \geq 0$ y sustituciones $(\sigma_1, \sigma_2) \in \mathcal{G}_k[\![\Gamma]\!]$, es $\left(\sigma_1\left(t_i^{U_i}\right), \sigma_2\left(t_i'\right)\right) \in \mathcal{T}_k[\![U_i]\!]$.

  Finalmente, aplicando las sustituciones σ_1 y σ_2 para $\widetilde{t}$ y t' respectivamente, se tiene el par $\left(\sigma_1([\ell_i = \widetilde{t_i}^{\,i\in 1..n}]^{U_i^{i\in 1..n}}), \sigma_2([\ell_i = {t'}_i^{\,i\in 1..n}])\right)$, lo cual, para demostrar que pertenece a la relación $\mathcal{T}_k[\![\mathcal{U}]\!]$ usamos la definición correspondiente de registros de la Figura 4.17 como sigue:
  Se puede considerar $U' = [\ell_i : {U''}_i^{\,i\in 1..n}]$ tal que $\varepsilon \vdash [\ell_i : {U_i}^{\,i\in 1..n}] \sim [\ell_i : {U''}_i^{\,i\in 1..n}]$ y $k' \leq k$, para probar que $\left(\varepsilon\sigma_1\widetilde{t} \bullet^{[\ell_i : {U_i}^{\,i\in 1..n}]} \ell_j, \langle U' \overset{\varepsilon}{\Longleftarrow} [\ell_i : {U_i}^{\,i\in 1..n}]\rangle\sigma_2 t'.\ell_j\right) \in \mathcal{T}_{k'}[\![U_j'']\!]$.
  Analizando la reducción de tales registros: si cualquier término se reduce a un error en menos de k' pasos, esto es por la función *iproj* o bien $\widetilde{proj}$ está indefinida, entonces el resultado se mantiene inmediatamente. El caso interesante es si éstos se reducen a valores relacionados en menos de k pasos. Entonces existen dos casos posibles para el campo j-ésimo.

  Por un lado, ambos son valores de registros (*records values*): $\widetilde{t}$ es un valor simple donde $v^U = [\ell_i = {u_i}^{\,i\in 1..n}]$ y $v' = [\ell_i = {u'}_i^{\,i\in 1..n}]$ cuyas proyecciones correspondientes dan $\varepsilon[\ell_i = {u_i}^{\,i\in 1..n}]^U.\ell_j \to iproj(\varepsilon, \ell_j)u_j :: U_j$ y $\langle U' \overset{\varepsilon}{\Longleftarrow} [\ell_i : {U_i}^{\,i\in 1..n}]\rangle v'.\ell_j \to \langle\langle U_j' \overset{iproj(\varepsilon,\ell_j)}{\Longleftarrow} U_j\rangle\rangle(v.\ell_j)$ los cuales están relacionados en $\mathcal{T}_{k'}[\![U_j]\!]$. Dada la Hipótesis de Inducción y siguiendo la definición en la Figura 4.17.

Por otro lado, $\widetilde{t}$ es un valor adscrito $v = \varepsilon[\ell_i = {v_i}^{i\in 1..n}] :: [\ell_i : {U_i}^{i\in 1..n}]$ cuya j-ésima proyección da la $iproj(\varepsilon', \ell_j)v_j :: U_j$ donde ε' es la composición de las evidencias para tal campo. Este valor junto con $\langle\langle U' \overset{iproj(\varepsilon,\ell_j)}{\Longleftarrow} [\ell_i : {U_i}^{i\in 1..n}]\rangle\rangle u.\ell_j$ están relacionados en $\mathcal{T}_{k'}[\![U_j]\!]$ siguiendo la definición de la Figura 4.17.

- **Caso** (U_{proj})
  Considérese $\widetilde{t}.\ell_j$ representado como $\check{t} \in TERM_{U_j}$. Esto significa que para la regla (U_{proj}), la hipótesis $\Gamma \vdash \widetilde{t}.\ell_j : U_j$ se mantiene debido a $\widetilde{proj}(U, \ell_j) = U_j$ y, también por la regla (IU_{proj}), $t^U \in TERM_U$ se mantiene con $\varepsilon \vdash U \sim [\ell_i : {U_i}^{i\in 1..n}]$.
  Siguiendo la regla (C_{proj}), el término compilado es t' en $\Gamma \vdash \widetilde{t}.\ell_j \Rightarrow \langle[\ell_i : {U_i}^{i\in 1..n}] \Longleftarrow U\rangle t'.\ell_j : \widetilde{proj}(U, \ell_j)$ cuyas hipótesis son $\Gamma \vdash \widetilde{t} \Rightarrow t' : U$ donde $U \sim [\ell_i = {U_i}^{i\in 1..n}]$.

  Supóngase $k \geq 0$ y $(\sigma_1, \sigma_2) \in \mathcal{G}_k[\![\Gamma]\!]$. Entonces, se quiere demostrar que $\left(\sigma_1\left(\widetilde{t}.\ell_j\right), \sigma_2\left(t'.\ell_j\right)\right) \in \mathcal{T}_k[\![U_j]\!]$ lo cual es $\left(\sigma_1\varepsilon\left(t^U \bullet^{[\ell_i:{U_i}^{i\in 1..n}]} \ell_j\right), \sigma_2\langle U \overset{\varepsilon}{\Longleftarrow} [\ell_i : {U_i}^{i\in 1..n}]\rangle(t'.\ell_j)\right) \in \mathcal{T}_k[\![U_j]\!]$

  Usando las hipótesis anteriores, la Hipótesis de Inducción se mantiene para $\widetilde{t}$ y t', para toda $k' \geq 0$ y las sustituciones $(\sigma_1, \sigma_2) \in \mathcal{G}_{k'}[\![\Gamma]\!]$, que es $\left(\sigma_1\left(\widetilde{t}\right), \sigma_2\left(t'\right)\right) \in \mathcal{T}_{k'}[\![U_1]\!]$. Este par, bajo el Lema 4.8.1 y el Lema 4.8.3 con $k' < k$ es justamente el caso anterior (U_{rec}).

■

Resumen

En este capítulo se presenta la propuesta completa con registros para tipos unión usando la metodología de Toro y Tanter [95]. Se realiza la extensión para los sistemas $STFL$, $GTFL$ y $GTFL_{\oplus}$ demostrando cada una de las propiedades, lemas y características requeridas tanto de manera estática como dinámica. En particular se prueban de manera formal tanto la propiedad de seguridad como la de consistencia. En el último lenguaje se realizan las demostraciones correspondientes para obtener la equivalencia entre las referencias basadas en evidencias y la semántica de traducción, para así proponer y demostrar a su vez la operación de conversión de tipos y la relación lógica basada en el Cálculo *Threesome*.

Capítulo 5

Conclusiones

> "Los retos más grandes que enfrentan los seres humanos en sus vidas son: (1) el reto de saber dónde empezar y (2) el reto de saber cuándo parar."
>
> *Sameh Elsayed.*

Se han desarrollado muchos enfoques de investigaciones en torno a la Tipificación Gradual desde diferentes perspectivas. En particular, los relacionados con registros y/o algún tipo de unión: Tipificado Gradual Abstracto usando subtipificado [42]; el estudio de la interacción entre la Tipificación Gradual con la unión y tipos de intersección [21]; uniones recursivas con tipos graduales [80]; refinamiento de sumas [52]; entre otros. Sin embargo, Toro y Tanter son los primeros en proponer una metodología para combinar el enfoque clásico de tipos graduales y los tipos unión para dar una interpretación gradual [95]. La tipificación gradual con uniones preserva las propiedades de seguridad y consistencia del lenguaje gradual a través de un enfoque estratificado de la tipificación gradual basado en una interpretación abstracta.

En este trabajo de tesis se presenta de forma detallada el uso del enfoque de Toro y Tanter utilizando un lenguaje con registros. Las propiedades de seguridad y consistencia (estáticas y dinámicas) para cada lenguaje utilizados en la metodología se demuestran, junto con las garantías estáticas y dinámicas del lenguaje $\mathsf{GTFL}^{\oplus}_{rec}$. La principal contribución de este trabajo es enriquecer un lenguaje gradual con una estructura de datos útil para los programadores. Esto es especialmente importante en un lenguaje funcional, donde las extensiones conservadoras del mismo, junto con otras estructuras de datos y constructores de tipos son deseables para el diseño de lenguajes de programación. Además, incluir registros en un lenguaje de programación permite extenderlo para comprender las nociones de construcciones similares, por ejemplo el tipo variante.

A partir de la revisión de las definiciones, proposiciones y lemas presentados por [95], algunos de ellos se ajustan en esta propuesta para integrar la estructura de registros. El objetivo es presentar un lenguaje gradual consistente y seguro siguiendo las ideas principales de los autores de esta metodología.

Es importante mencionar que el enfoque de registros que aquí se presenta difiere del presentado en [42], donde se introduce un tipo gradual distinto, denominado **fila gradual** (*gradual row*). Una fila gradual se expresa mediante campos adicionales con el tipo ? en el mismo registro. Estos se representan de la forma $[label_1 : type_1, label_2 : type_2, ..., label_n : type_n, ?]$, donde cada etiqueta (`label`) hace referencia a un identificador de cada campo del registro, el cual tiene asociado un tipo correspondiente (`type`). Los registros con filas graduales representan cualquier tipo de registro que tenga al menos las etiquetas enumeradas con algún tipo correspondiente, y el tipo ? [42].

En este trabajo se utiliza el criterio de corrección para la tipificación gradual que Cimini et al. en [25] denomina *extensión conservadora*, el cual se define como: para todo $\Gamma, e\ y\ T, \Gamma \vdash e : T$ si y sólo si $\Gamma \vdash_G e : T$. Por un lado, se puede decir que la extensión conservadora garantiza que los programas bien tipificados del lenguaje original siguen estando bien tipificados en el lenguaje con tipificado gradual. Por otro lado, los programas mal tipificados en los lenguajes originales seguirán estando mal tipificados en un lenguaje con tipificado gradual.

A lo largo de este trabajo de tesis se desarrollan varios sistemas para que un lenguaje gradual pueda extenderse con registros de manera consistente y segura. Lo anterior implica demostrar que en cada sistema se preserven las propiedades estáticas y dinámicas al añadir registros, es decir, tanto en el lenguaje $STFL$, como en $GTFL$ y $GTFL_{\oplus}$. En particular, en este último lenguaje se debe preservar la consistencia y seguridad de tipos al añadir la semántica traslacional, misma que sirve para comprobar las propiedades asociadas a la conversión de tipos y la relación lógica basada en la compilación del Cálculo *Threesome*. En particular, en tiempo de ejecución se puede enviar un error al programador, si la operación de conversión de tipos no se puede resolver. Este `error` se define no como un tipo, sino como parte del comportamiento del lenguaje. Basándose en la metodología en la cual se desarrolla este trabajo de tesis, el sistema de tipos acepta y rechaza los mismos programas tipificados estáticamente, pero acepta más programas tipificados gradualmente [42].

Dado lo anterior se puede decir que la hipótesis de este trabajo es cierta, y ha sido demostrada paso a paso para cada uno de los sistemas aquí presentados, es decir, que para todo programa en un lenguaje diseñado bajo un sistema de tipos gradual al extenderlo con registros usando el enfoque de tipos unión se mantiene su consistencia.

Resumen de las contribuciones

Al inicio de esta tesis se presenta un resumen de las diferentes propuestas realizadas hasta el momento acerca de los distintos enfoques de tipificado en lenguajes de programación, obteniendo una propuesta en donde convergen todas las posturas que se han investigado a través de los años en el área. Esta propuesta puede ser de ayuda en cursos relacionados con lenguajes de programación para ver de manera sintetizada los diferentes enfoques, lo cual ayuda a que futuros investigadores puedan tener en un solo lugar este tipo de información ya analizada y sintetizada.

Las principales aportaciones de este estudio son (1) el analizar, y completar la teoría de la propuesta expuesta por [95] utilizando una estructura de datos como registros en un lenguaje de tipificación gradual. (2) El estudio de una metodología a través de una extensión utilizando una estructura de datos como los registros, permite reflexionar sobre los conceptos más importantes relacionados tanto en la metodología como en la propia estructura de datos.

El enfoque dado por [95] permite utilizar la metodología AGT. Por un lado, la semántica de $\mathsf{GTFL}^{\oplus}_{rec}$ puede derivarse sistemáticamente. Por otro lado, siguiendo el marco de interpretación abstracta podemos demostrar tanto la seguridad como la garantía gradual con la extensión de los registros. Al utilizar la compilación del Cálculo *Threesome* junto con la semántica de un lenguaje interno es equivalente a la semántica operativa derivada con AGT. Por último, utilizando relaciones lógicas se demuestra la corrección de sus argumentos.

Cabe resaltar que se analizaron y modificaron algunas de las definiciones y proposiciones hechas en el trabajo propuesto por Toro y Tanter en [95], así como la corrección de errores del reporte técnico extendido [96] en el cual se sustentan y desarrollan las demostraciones del artículo original. En este trabajo doctoral se corrijen los errores encontrados en los artículos anteriores, pues se encontraban mal presentados, para así lograr la consistencia y seguridad del lenguaje extendido con registros. A su vez, se propusieron algunas definiciones para esclarecer el contenido del trabajo, estas definiciones propuestas no se encontraban expuestas en el trabajo original de Toro y Tanter.

Otra de las contribuciones que se deben mencionar es el hecho de que en este trabajo se explica a detalle y con ejemplos la metodología propuesta por los autores del trabajo original.

Investigación a futuro

Un tipo de investigación que se pudiera derivar de este trabajo es la extensión con este mismo tipo de metodología para el caso de tipos inductivos, teniendo como caso base el uso de registros. Asimismo, se podría extender esta investigación usando polimorfismo, o bien extender el sistema con objetos, donde un objeto podría tratarse como un registro más complejo[1] pues un registro podría hacer recursión o llamadas a algún campo a través de la

[1] Coloquialmente podría decirse que sería un registro con esteroides.

primitiva *this*[2].

En el año 2019, [94] presenta la tipificación gradual con polimorfismo paramétrico explícito. Este estudio muestra que la parametricidad es incompatible con la garantía gradual dinámica. Otra investigación de investigación presentada en [51] muestra una extensión del Sistema F, denominada Sistema F_G. En este estudio se presenta un lenguaje de tipificación gradual con polimorfismo paramétrico, demostrando la propiedad de seguridad de tipos, la garantía gradual, y el teorema de subtipificado de la culpa (*blame subtyping*), que se conoce como "criterio refinado".

La implementación de la unión gradual ($\oplus$) para el uso de registros puede ser hecha sobre todas las etiquetas un registro, es decir, introducir la unión gradual sobre cada etiqueta que contenga un registro, o bien se podría hacer una optimización de ésta solo introduciendo la unión gradual en la etiqueta del registro sobre la cual se realizará la projección del registro. La segunda opción implicaría el haber revisado la etiqueta sobre la cual se va a hacer la proyección y verificar que dicha etiqueta exista en la definición del registro. De la primera propuesta, no se tendría que hacer dicha verificación pues el introducir la unión gradual sobre todas las etiquetas no requeriría hacer primero la búsqueda sobre la etiqueta escrita en la proyeccción, sin embargo en manejo de espacio esto requeriría mayor espacio en memoria para almacenar todas las uniones graduales sobre cada etiqueta.

Finalmente, es importante recordar que este enfoque muestra un estudio completo relacionado con un lenguaje de tipificación gradual ampliado con registros, para mantener su consistencia, y seguridad a través del análisis de tipificado estático y dinámico, preservando a su vez la seguridad y consistencia en todos los sistemas utilizados en este enfoque estratificado.

Publicaciones derivadas de este trabajo doctoral

Derivado de este trabajo de tesis doctoral se han realizado las siguientes publicaciones, hasta el momento en que se presenta éste:

- Ramírez Pulido Karla, Ortega-Arjona, Jorge Luis, y González Huesca Lourdes del Carmen (2020). Gradual typing using union typing with records. Electronic Notes in Theoretical Computer Science, 354, 171-186. ELSEVIER, ISSN $1571 + 0661$, $https://doi.org/10{,}1016/j.entcs{,}2020{,}10{,}013$.
- Ramírez Pulido Karla, Ortega-Arjona, Jorge Luis, y González Huesca Lourdes del Carmen (2019). Gradual typing using union typing with records. LANMR 2019 Proceedings of the Twelfth Latin American Workshop on Logic/Languages, Algorithms and New Methods of Reasoning, CEUR Workshop Proceedings. Vol+2585, $urn:nbn:de:0074+2585+4http://ceur+ws.org/Vol+2585/paper3.pdf$.

[2]Usando el *argot* de la P.O.O.

Bibliografía

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. ACM transactions on programming languages and systems (TOPLAS), 13(2):237–268, 1991.

[2] Samson Abramsky and Chris Hankin. An introduction to abstract interpretation. In Abstract Interpretation of declarative languages, volume 1, pages 63–102. Ellis Horwood, 1987.

[3] Esteban Allende, Johan Fabry, and Éric Tanter. Cast insertion strategies for gradually-typed objects. In Proceedings of the 9th Symposium on Dynamic Languages, DLS 13, pages 27–36, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Gregory R. Andrews. Concurrent programming: principles and practice. Benjamin/Cummings Publishing Company, 1991.

[5] Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '16, pages 49–60, New York, NY, USA, 2016. ACM.

[6] Andrew W. Appel and P. Jens. Modern Compiler Implementation in Java. Cambridge University Press, 1era. edition, 1998.

[7] Jorge Luis Arjona Ortega. Software: Tecnología para el procesamiento de información. `http://lya.fciencias.unam.mx/jloa/opinion2.html`, 2000. Consultado el 18 de octubre de 2016.

[8] Robert Atkey. The syntax and semantics of quantitative type theory. Proc. ACM Program. Lang. Art.1, (1), enero 2017.

[9] Arthur Baars and S. Doaitse Swierstra. Typing dynamic typing. In ACM SIGPLAN Notices, volume 37, pages 157–166. ACM, octubre 2002.

[10] Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In International Conference on Verification, Model Checking, and Abstract Interpretation, pages 25–46. Springer, 2018.

[11] Felipe Bañados Schwerter, Alison M Clark, Khurram A Jafery, and Ronald Garcia. Abstracting gradual typing moving forward: precise and space-efficient. Proceedings of the ACM on Programming Languages, 5(POPL):1–28, 2021.

[12] Felipe Banados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, pages 283–295, 2014.

[13] Felipe Andrés Bañados Schwerter. Gradual typing for generic type-and-effect systems. 2014.

[14] Alexis Beingessner and Steve Klabnik. The rustonomicon: The dark arts of advanced and unsafe rust programming, 2016. Consultado el 22 de febrero de 2022.

[15] Derek Bronish. Abstraction as the Key to Programming, with Issues for Software Verification in Functional Languages. Dissertation Thesis Degree: PhD. PhD thesis, The Ohio State University, 2012.

[16] F. Brooks and H.J. Kugler. No silver bullet. Abril, 1987.

[17] Frank. Buschman, Rregine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented. Software Architecture. A system of Patterns, volume 1. Wiley New York, 2004.

[18] Robert Cartwright and Mike Fagan. Soft typing. In ACM SIGPLAN Notices, volume 26, pages 278–292. ACM, 1991.

[19] Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. Proceedings of the ACM on Programming Languages, 1(ICFP):41, 2017.

[20] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. Gradual typing: a new perspective. Proceedings of the ACM on Programming Languages, 3(POPL):1–32, 2019.

[21] Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. Set-theoretic types for polymorphic variants. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pages 378–391, New York, NY, USA, 2016. ACM.

[22] David R. Cheriton. Interpreter based program language translator using embedded interpreter types and variables. `https://www.google.com/patents/US9262135`, febrero 2016. US Patent 9,262,135.

[23] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. In ACM SIGPLAN Notices, volume 34, pages 62–72. ACM, 2000.

[24] Alonzo Church. A formulation of the simple theory of types. The journal of symbolic logic, 5(2):56–68, 1940.

[25] Matteo Cimini and Jeremy G Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. ACM SIGPLAN Notices, 51(1):443–455, 2016.

[26] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In European Symposium on Programming, pages 520–535. Springer, 2007.

[27] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent Types for Low-Level Programming, pages 520–535. Berlin, Heidelberg, 2007.

[28] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. Ccured in the real world. In ACM SIGPLAN Notices, volume 38, pages 232–244. ACM, 2003.

[29] Patrick Cousot. Abstract interpretation. ACM Comput. Surv., 28(2):324–328, junio 1996.

[30] Patrick Cousot. Types as abstract interpretations. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 316–331. ACM, 1997.

[31] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In Proceedings of the seventh international conference on Functional programming languages and computer architecture, pages 170–181. ACM, 1995.

[32] Martin D. Davis and Weyuker Elaine J. Computability, Complexity and Languages. Fundamentals of Theoretical Computer Science. Academic Press, Inc., 1era. edition, 1986.

[33] Tim Disney and Cormac Flanagan. Gradual information flow typing. In International workshop on scripts to programs, 2011.

[34] Matthias Felleisen. Adding types to untyped languages. In Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation, pages 1–2. ACM, enero 2010.

[35] Matthias Felleisen, David Van Horn, Conrad Barski, et al. Realm of Racket: Learn to Program, One Game at a Time! No Starch Press, 2013.

[36] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In ACM SIGPLAN Notices, volume 37, pages 48–59. ACM, 2002.

[37] Cormac Flanagan. Hybrid type checking. In Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 245–256, 2006.

[38] Daniel P. Friedman, Mitchell Wand, and Christopher Thomas Haynes. Essentials of programming languages. MIT press, 2001.

[39] Miguel Garcia and Francisco Ortin. Optimización de lenguajes con comprobación estática y dinámica de tipos. 2012.

[40] Miguel Garcia, Francisco Ortin, and Jose Quiroga. Design and implementation of an efficient hybrid dynamic and static typing language. Software: Practice and Experience, 46(2):199–226, 2016.

[41] Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. ACM SIGPLAN Notices, 50(1):303–315, 2015.

[42] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM.

[43] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. ACM Transactions on Programming Languages and Systems (TOPLAS), 36(4):12, 2014.

[44] Ben Greenman. Soft typing, abril 2017. Notas de clase.

[45] Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In Trends in Functional Programming, pages 54–70. Citeseer, 2007.

[46] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In Scheme and Functional Programming Workshop, volume 6, pages 93–104, 2006.

[47] Robert Harper. Practical foundations for programming languages. Cambridge University Press, 2da. edition, 2016.

[48] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. Higher-Order and Symbolic Computation, 23(2):167, 2010.

[49] C. A. R. Hoare. Procedures and parameters: An axiomatic approach.

[50] C.A.R. Hoare. A theory of programming: Denotational, algebraic and operational semantics, 1999.

[51] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On polymorphic gradual typing. Proceedings of the ACM on Programming Languages, 1(ICFP):40, 2017.

[52] Khurram A. Jafery and Joshua Dunfield. Sums of uncertainty: Refinements go gradual. ACM SIGPLAN Notices, 52(1):804–817, 2017.

[53] Ranjit Jhala, Eric Seidel, and Vazou Niki. Programming with refinement types. https://ucsd-progsys.github.io/liquidhaskell-tutorial. Consultado el 22 de febrero de 2022.

[54] Delia Kesner and Pierre Vial. Types as resources for classical natural deduction. In LIPIcs-Leibniz International Proceedings in Informatics, volume 84. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[55] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. ACM Transactions on Programming Languages and Systems (TOPLAS), 32(2):6, 2010.

[56] Shriram Krishnamurthi. Programming Languages. Application and Interpretation. Brown University, 2da. edition, 2007.

[57] Shriram Krishnamurthi. Programming languages: Application and interpretation. Brown University, 2012.

[58] Thomas Kühne. Unifying nominal and structural typing. Software & Systems Modeling, pages 1–15, 2018.

[59] Elizabeth Labrada, Matías Toro, and Éric Tanter. Gradual system f. arXiv preprint arXiv 1807.04596, 2018.

[60] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. OOPSLA, 2004.

[61] Nevena Milojković and Oscar Nierstrasz. Exploring cheap type inference heuristics in dynamically typed languages. In Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, pages 43–56, New York, NY, USA, 2016. ACM.

[62] Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. Proceedings of the ACM on Programming Languages, 1(OOPSLA):1–30, 2017.

[63] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. Principles of program analysis. Springer, 1999.

[64] John K. Ousterhout. Scripting: Higher level programming for the 21st century. Computer, 31(3):23–30, 1998.

[65] Frank Pfenning. Church and curry: Combining intrinsic and extrinsic typing. Studies in Logic and the Foundations of Mathematics, 2008.

[66] Benjamin C. Pierce. Types and programming languages. MIT press, 2002.

[67] Gordon D. Plotkin. The origins of structural operational semantics. The Journal of Logic and Algebraic Programming, 60:3–15, 2004.

[68] Terrence W. Pratt, Marvin V. Zelkowitz, and Tadepalli V. Gopal. Programming languages: design and implementation. Prentice-Hall, Englewood Cliffs, 3era. edition, 1998.

[69] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. ACM SIGPLAN Notices, 47(1):481–494, 2012.

[70] Brianna M. Ren and Jeffrey S. Foster. Just-in-time static type checking for dynamic languages. ACM SIGPLAN Notices, 51(6):462–476, 2016.

[71] Felipe Bañados Schwerter. Side effects take the blame. In SLE, pages 195–206, 2016.

[72] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. Gradual type-and-effect systems. Journal of functional programming, 26, 2016.

[73] Stewart Shapiro. Classical logic. `http://plato.stanford.edu/archives/win2009/entries/logic-classical/`, 2009. Consultado el 13 de octubre de 2016.

[74] Jeremy Siek. What is a gradual typing. Indiana University Bloomington `http://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/`, 2014. Consultado el 25 de agosto de 2016.

[75] Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In Giuseppe Castagna, editor, Programming Languages and Systems, pages 17–31, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[76] Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In European Symposium on Programming, pages 17–31. Springer, 2009.

[77] Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercion: Together again for the first time. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 15, pages 425 – 435, New York, NY, USA, 2015. Association for Computing Machinery.

[78] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In Scheme and Functional Programming Workshop, volume 6, pages 81–92, 2006.

[79] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In European Conference on Object-Oriented Programming, pages 2–27. Springer, 2007.

[80] Jeremy G. Siek and Sam Tobin-Hochstadt. The recursive union of some gradual types. In A List of Successes That Can Change the World, pages 388–410. Springer, 2016.

[81] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In Proceedings of the 2008 symposium on Dynamic languages, page 7. ACM, 2008.

[82] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In 1st Summit on Advances in Programming Languages (SNAPL 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[83] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In European Symposium on Programming Languages and Systems, pages 432–456. Springer, 2015.

[84] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '10, pages 365–376, New York, NY, USA, 2010. ACM.

[85] Brian Siritzky. Executable operational semantics of programming languages. Dissertation Thesis Degree: PhD. PhD thesis, New York University, 1993.

[86] Gideon Joachim Smeding. An executable operational semantics for Python. Dissertation Thesis Degree: PhD. PhD thesis, Universiteit Utrecht, 2009.

[87] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In ACM SIGPLAN Notices, Vol. 51, No. 1, pages 456–468. ACM, enero 2016.

[88] Asumu Takikawa, T Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In ACM SIGPLAN Notices, volume 47, pages 793–810. ACM, 2012.

[89] Ross Tate. Retargeting gradual typing (invited talk). In LIPIcs-Leibniz International Proceedings in Informatics, volume 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[90] Satish Thatte. Quasi-static typing. In Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 367–381. ACM, 1989.

[91] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 964–974, 2006.

[92] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. ACM SIGPLAN Notices, 43(1):395–406, 2008.

[93] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory typing: Ten years later. In LIPIcs-Leibniz International Proceedings in Informatics, volume 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[94] Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. Proceedings of the ACM on Programming Languages, 3(POPL):1–30, 2019.

[95] Matías Toro and Éric Tanter. A gradual interpretation of union types. In International Static Analysis Symposium, pages 382–404. Springer, 2017.

[96] Matías Toro and Éric Tanter. Gradual union types, complete definition and proofs. Technical Report TR/DCC-2017-1, University of Chile, June 2017.

[97] Autores varios. Sitio oficial de citas textuales en el Web. Consultado el 17 de octubre de 2017.

[98] Autores varios. Sitio oficial de citas textuales en el Web. Consultado el 17 de diciembre de 2021.

[99] Autores varios. Sitio oficial de StackExchange. https://cs.stackexchange.com/questions/93798/ Consultado el 16 de febrero de 2022.

[100] Autores varios. Sitio oficial de StackOverflow. https://stackoverflow.com/questions/9338709/what-is-dependent-typing. Consultado el 22 de febrero de 2022.

[101] Autores varios. Sitio oficial del lenguaje de programación SAGE. `https://sage.soe.ucsc.edu/`. https://sage.soe.ucsc.edu/ Consultado el 2 de noviembre de 2021.

[102] Niki Vazou, Patrick M Rondon, and Ranjit Jhala. Abstract refinement types. In European Symposium on Programming, pages 209–228. Springer, 2013.

[103] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract Refinement Types, pages 209–228. Berlin, Heidelberg, 2013.

[104] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In ACM SIGPLAN Notices, Vol. 50, No. 2, pages 45–56. ACM, octubre 2014.

[105] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: open-world soundness and collaborative blame for gradual type systems. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pages 762–774. ACM, 2017.

[106] Niklaus Wirth. A plea for lean software. Computer, 28(2):64–68, 1995.

[107] Ningning Xie, Xuan Bi, Bruno CDS Oliveira, and Tom Schrijvers. Consistent subtyping for all. ACM Transactions on Programming Languages and Systems (TOPLAS), 42(1):1–79, 2019.