Compilación de Programas Seguros

Enrique Molinari

Director: Dr. Eduardo Bonelli
Codirector: Dr. Gustavo Rossi

Tesis presentada para obtener el grado de Magíster en Ingeniería de Software

Facultad de Informática, Universidad Nacional de La Plata

Diciembre 2009

Índice general

Índice de figuras

Capítulo 1

Introducción

Garantizar la confidencialidad de los datos en los sistemas de información es una cuestión frecuentemente abordada por investigadores. El objetivo final de esta búsqueda se relaciona con garantizar que la información sensible sea compartida exclusivamente entre partes autorizadas. Para esto es necesario tener control sobre la forma en que dicha información se propaga dentro del sistema, y evitar que sea accedida por partes no autorizadas.

El volumen de datos que manejan los sistemas de información es cada vez mayor y hasta el momento no existe una única solución capaz de garantizar que los datos confidenciales sean correctamente protegidos. En los últimos años la creciente interacción entre sistemas de software que permiten a sus usuarios descargar código desde terceras partes y ejecutarlo localmente, ha significado nuevos y mayores desafíos para los investigadores. Cómo garantizar al receptor que la ejecución local de código descargado no representa una amenaza sobre la integridad y confidencialidad de sus datos, es un problema en permanente discusión, que intenta ser superado desde distintos enfoques.

La utilización de técnicas basadas en lenguajes de programación (ver sección 1.1), las cuales desarrollaremos a lo largo de este trabajo, que permitan proteger la confidencialidad de los datos, aparece como una solución prometedora. En esta nueva era donde es natural descargar software foráneo para luego ejecutarlo localmente, los mecanismos de seguridad convencionales, generalmente basados en capturar las llamadas al sistema no son suficientes y es aquí donde las técnicas basadas en lenguajes de programación toman protagonismo.

Uno de los aspectos que se observa al indagar sobre el rumbo que ha tomado la investigación de esta problemática, es el hecho de que existe mucha bibliografía sobre el control de flujo de información (ver sección 1.4) utilizando técnicas de lenguajes de programación, casi la totalidad de la misma está orientada a lenguajes de alto nivel. Esto es llamativo porque por más precisas que sean estas técnicas lo que termina ejecutándose es la salida del compilador, o sea, un lenguaje de máquina o más cercano al lenguaje de máquina. Si consideramos que el código móvil, tal es la denominación que recibe

el software descargado para luego ser ejecutado localmente, se encuentra mayormente como lenguaje de bajo nivel [SM03] resulta imperativo aplicar los mecanismos de control de flujo sobre lenguajes de bajo nivel. En esta dirección, se han presentado algunos trabajos como los desarrollados en [YI06], [YU07], [BCM04, MED06].

En particular en [BCM04, MED06], se presenta un lenguaje tipo assembler llamado SIFTAL con control de flujo de información. Los autores utilizan técnicas estáticas que permiten verificar en tiempo de compilación si el programa escrito en SIFTAL no viola las políticas de confidencialidad.

Este primer capítulo intenta dar una introducción a las técnicas basadas en lenguajes de programación para el control del flujo de información. Presenta un resumen sobre las diferentes líneas de investigación y los trabajos relacionados. Finalizando el capítulo con la presentación de la estructura de éste trabajo y sus contribuciones.

1.1. Seguridad Basada en Lenguajes de Programación

En los 70 los sistemas estaban aislados. Para llegar a ellos era necesario pasar barreras de seguridad físicas. Las actualizaciones de software eran realizadas por expertos administradores. Los sistemas operativos estaban implementados con relativamente pocas líneas de código y daban soporte a un conjunto reducido de aplicaciones. Las políticas de seguridad estaban basadas en términos de los objetos que el sistema operativo manejaba, como los procesos, archivos, dispositivos periféricos, etc.

Internet y los sistemas distribuidos cambiaron totalmente este entorno. Los sistemas de software ya no están aislados y son constantemente actualizados incluso muchas veces sin el consentimiento del usuario. Todo es ejecutable, un email, una página web, etc. Los sistemas operativos aplican las políticas de seguridad cuando existen llamadas al sistema y sobre los objetos que maneja, procesos, archivos, etc, tratándolos como caja negra. Debido a estas dos cuestiones se hace difícil que el sistema operativo pueda controlar las aplicaciones que reposan sobre él cuando no realizan llamadas al sistema [MO01].

Hoy, los sistemas operativos no deben ni pueden encargarse de garantizar la seguridad a nivel aplicación, mas aún en entornos distribuidos donde el código móvil es uno de los principales generadores de inseguridad [MO01]. Los mecanismos de control de seguridad basados en lenguajes de programación surgen como una solución prometedora. Éstos se basan en analizar el código de forma de detectar posibles violaciones a cierta política de seguridad. Un intérprete de cierto lenguaje podría incluir chequeos de seguridad tan o más eficaces como los que podríamos encontrar en los chequeos realizados a nivel del kernel de un sistema operativo. Dado que estos chequeos en ejecución podrían afectar considerablemente la performance del programa, también es posible moverlos al compilador. Un chequeador de tipos podría ser utilizado para deducir, en tiempo de compilación, cómo se va a comportar el programa en ejecución y de esta

forma eliminar chequeos realizados en tiempo de ejecución. Este trabajo sigue la línea que propone utilizar técnicas en tiempo de compilación para detectar violaciones sobre políticas de seguridad previamente definidas.

1.2. Código Móvil y Proof Carrying Code

El software descargado para luego ser ejecutado localmente, dentro de un entorno de sistemas distribuidos, se denomina *código móvil*. Denominaremos al proveedor de código móvil *productor*, y *consumidor* a quien descarga dicho código para ejecutarlo localmente.

La interacción entre sistemas por medio de código móvil presenta una poderosa forma de extender la funcionalidad de los sistemas de software, pero al mismo tiempo compromete la seguridad del consumidor. De aquí surgen las siguientes cuestiones:

- ¿Cómo podemos garantizar que la ejecución del código móvil no utilizará recursos del consumidor en forma malintencionada?
- ¿Cómo podemos lograrlo sin la necesidad de una compleja infraestructura y sin afectar la performance de su ejecución?

Diferentes iniciativas han surgido para dar solución a estos problemas, entre las que se encuentran [NEC01]:

- Firma digital

  Esta técnica le garantiza al consumidor que el código a descargar fue producido por una organización o persona en quien el consumidor podrá confiar o no. Aunque exista la confianza entre el productor y el consumidor, esta técnica es débil, dado que el código podría ser igualmente malicioso debido a errores de programación o desconocimiento.

- Monitoreo de la ejecución

  Se basa en monitorear la ejecución del código descargado y finalizarla si se intenta realizar alguna acción que no esta permitida. Por ejemplo, los applets de Java se ejecutan en un ambiente en el cual se monitorea su ejecución. En cuanto realizan alguna acción que viola alguna política de seguridad (como por ejemplo leer o escribir en el disco local) el entorno donde son ejecutados genera una excepción, abortando la ejecución. El principal problema que presenta este mecanismo de control es que afecta considerablemente la performance.

- Java Bytecode

  Consiste en utilizar Java Bytecode [GJS96] como código móvil. Es simple y con buena aceptación, pero con las siguientes desventajas:

- Específico para Java

- Se requiere el chequeador de tipos de Java y el intérprete instalados en el consumidor. Estructura demasiado grande para confiar.

- Como alternativa al intérprete se puede utilizar Java JIT (Just in Time) Compiler, que traduce bytecode a código nativo. Nuevamente el JIT compiler es una pieza compleja y grande para confiar y lo que es más importante, el código nativo que será finalmente ejecutado no ha sido verificado.

La aparición de Proof Carrying Code [NEC01] nace como alternativa a las técnicas tradicionales planteadas anteriormente. PCC requiere que el productor provea junto con el código a entregar una prueba formal que garantiza que se cumple con ciertas políticas de seguridad definidas por el consumidor. Luego, el consumidor debe verificar que la prueba es válida para el código entregado. La verificación de la prueba es simple y rápida a diferencia de la construcción de la prueba que requiere realizar el productor. Si la prueba es válida el consumidor se asegura que la ejecución será segura.

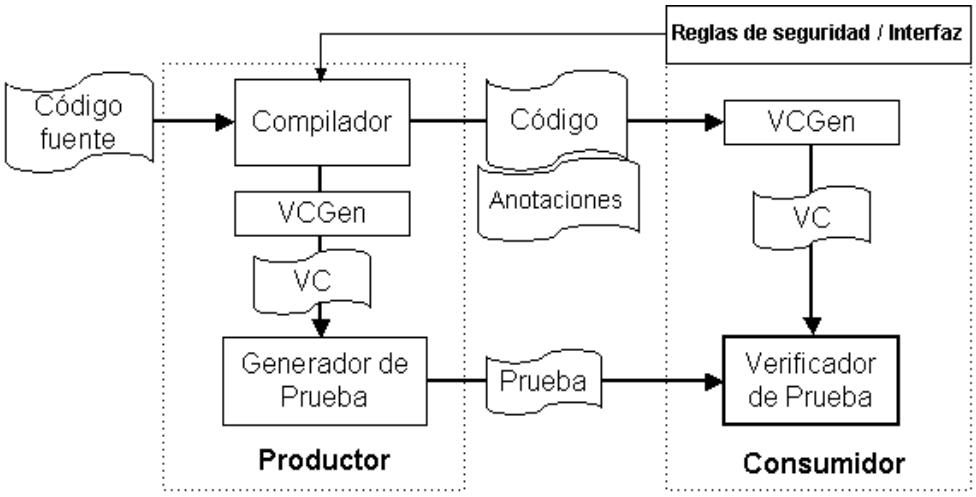

Figura 1.1: Arquitectura PCC

En la Figura 1.1 se presentan los dos actores principales de la arquitectura Proof Carrying Code: el *productor* y el *consumidor*. Cada actor posee diferentes responsabilidades y cuenta con diferentes componentes para ejecutarlas. Como punto de partida en el uso de la arquitectura, el consumidor debe definir la política de seguridad donde se especifica bajo qué condiciones es seguro ejecutar el código provisto por el productor. La política de seguridad está compuesta por las *reglas de seguridad* y la *interfaz*. Las reglas de seguridad describen todas las operaciones autorizadas y sus precondiciones de seguridad asociadas. La interfaz describe las convenciones de llamada entre el consumidor de código y el código remoto, estas son las invariantes que mantiene el consumidor cuando invoca el código remoto y las invariantes que el código remoto debe establecer antes de llamar a las funciones proporcionadas por el consumidor.

El productor de código debe compilar la pieza de software que desea compartir con el consumidor tomando como entrada además del código fuente, la política de seguridad. El resultado de la compilación es código de bajo nivel o intermedio, acompañado de *anotaciones*. Las *anotaciones* consisten en especificar los invariantes de los ciclos y un par compuesto por una precondición y una postcondición por cada función provista. La precondición de una función es una especificación formal de una condición que debe ser verdadera para poder invocar a la función. La postcondición de una función es una especificación formal de una condición que debe ser verdadera luego de finalizada la ejecución de la función. A este tipo de compiladores que generan código con anotaciones Necula en [NEC01] lo denomina *compilador certificante*.

El código de bajo nivel junto con las anotaciones embebidas en él es procesado por el *VCGen* (Verification Condition Generation) quién produce un predicado de seguridad para cada función del código, de manera tal que dichos predicados puedan ser demostrados si y sólo si el código a entregar es seguro de acuerdo a la política de seguridad. Luego, los predicados de seguridad forman la *condición de verificación* (VC), la cual es suministrada a un *generador de pruebas* quien intenta probar el predicado. Si se prueba con éxito la validez del predicado, se emite una representación de la prueba en LF (Logical Framework) [HAR1], que es esencialmente, cálculo lambda tipado.

Del lado del consumidor, el *verificador de prueba* recibe la condición de verificación, que se vuelve a generar analizando el código y sus anotaciones embebidas, y la prueba expresada en LF. Su responsabilidad es comprobar que la prueba obtenida demuestra la condición de verificación. Dado que la prueba es expresada en LF la verificación de la misma se puede llevar a cabo implementando un chequeador de tipos de LF.

A diferencia de los otros mecanismos, PCC posee las siguientes ventajas:

- No requiere una gran infraestructura, dado que la verificación de la prueba es simple.
- No afecta la performance de la ejecución dado que la verificación se realiza antes de la ejecución.
- No requiere de otros mecanismos para funcionar como la criptografía o autenticación, aunque pueden coexistir sin problemas. Si la prueba o el código es modificado la verificación fallará. Si el código modificado junto con la prueba modificada pasan la verificación, entonces la ejecución igualmente será segura. Esto es conocido como *Tamper resistance*.
- Se puede utilizar para validar muchas propiedades de seguridad; básicamente, si se puede probar su validez, PCC lo puede verificar.

1.3. Proof Carrying Code y Preservación de Tipos en Compilación

Para que la idea de PCC pueda ser llevada a la práctica eficientemente, como se menciona en [PI01], necesitamos dar solución a dos problemas:

- ¿Qué propiedades de seguridad queremos que el código posea?
- ¿De qué forma los productores de código móvil van a construir la prueba formal que garantiza que el código que entregan posee las propiedades deseadas?

La primera pregunta depende del contexto en el que nos encontremos. En el contexto de este trabajo, queremos que cumpla con las políticas de confidencialidad de la información. Es decir que los datos identificados como confidenciales, no lleguen a personas no autorizadas. A esta propiedad se la conoce como *no-interferencia* (ver sección 1.6).

La segunda pregunta trae a la práctica la imposibilidad de generar automáticamente para cualquier porción de código la construcción de una prueba formal. Es muy costoso (y a veces irrealizable) para el productor de código generar la prueba. Este problema abrió varias líneas de investigación, que no serán discutidas aquí.

Dado que PCC se presenta como una arquitectura genérica cuya idea es permitir al consumidor chequear fácilmente que el código descargado posee ciertas propiedades de seguridad, puede ser implementado de varias formas. Una de estas formas, y quizás la más prometedora hoy, es combinar PCC con la técnica conocida como preservación de tipos en compilación (*type-preserving compilation* [MOR85]) [PI01]. La idea es expresar las propiedades de seguridad deseadas en términos de expresiones de tipos. De esta forma, los programadores actúan como probadores de teoremas, dado que el programa escrito en un lenguaje de alto nivel debe pasar el chequeo de tipos, en caso contrario, debe ser modificado hasta que la compilación se pueda realizar. Preservación de tipos en compilación, requiere que tanto el lenguaje de alto nivel como el lenguaje de bajo nivel sean tipados, dado que no sólo traducen las sentencias de alto nivel a instrucciones en un lenguaje de bajo nivel, sino que también traducen los tipos del lenguaje de alto nivel a tipos en el lenguaje de bajo nivel [MOR85]. Preservan los tipos de las expresiones, preservando, conceptualmente, la prueba. El código móvil se distribuye con sus anotaciones de tipos, y el consumidor verifica las propiedades de seguridad aplicando un chequeador de tipos.

En [WAL99] se presenta TAL (Typed Assembly Language). TAL es un lenguaje assembler al estilo RISC con un sistema de tipos que permite garantizar principalmente dos propiedades de seguridad:

1. Control del flujo, es decir que los saltos no se realicen hacia direcciones de memoria inválidas, sino siempre hacia entradas válidas del programa.

2. Que las operaciones se realicen sobre tipos abstractos de datos correctos. Por ejemplo, que no nos permita sumar (instrucción *add*) una dirección de memoria de inicio de un bloque de código con un valor entero.

De esta forma, en una arquitectura PCC un compilador con preservación de tipos podría generar código TAL y el consumidor como parte de la verificación de la prueba ejecutará el chequeador de tipos de TAL. Si el programa pasa el chequeo de tipos, podremos ejecutarlo con la garantía de que no violará las políticas de seguridad para las cuales TAL fue diseñado.

En el contexto de nuestro trabajo y dentro de la arquitectura PCC, la propiedad que queremos verificar es *no-interferencia*.

1.4. Seguridad en el Flujo de Información

Diferentes mecanismos de seguridad han sido utilizados para garantizar la confidencialidad de los datos, entre los cuales podemos mencionar la encriptación que permite proteger la información al transmitirla por una red. Los firewalls que protegen las comunicaciones internas de las que existen fuera de nuestro sistema. Los antivirus, que se basan en detectar patrones de código previamente marcados como malintencionados. Ninguno de ellos controla de qué forma se propaga la información.

En el mecanismo de control de acceso conocido como *Discretionary Access Control* (DAC) que fue diseñado para detectar y prevenir accesos no autorizados sobre objetos de un sistema, se identifica a cada sujeto autorizando o no el acceso a los objetos. Pero de la misma forma que los otros mecanismos, una vez concedido el acceso sobre el objeto, no se controla cómo ese objeto se propaga dentro del sistema. Es decir, que un sujeto (por ejemplo un usuario) con privilegios de administrador podría ejecutar código malintencionado o programas denominados troyanos los cuales al ejecutarse con privilegios de administrador tienen acceso a cualquier objeto del sistema pudiendo copiar datos sensibles en otros objetos que requieran menos privilegios de acceso. Como consecuencia, usuarios no autorizados podrían acceder a ellos.

Para garantizar que los datos sensibles son utilizados sin violar las políticas de confidencialidad no basta con restringir el acceso a ellos, sino que es necesario regular el *flujo* de dichos datos dentro del sistema.

Bell y LaPadula [BP73] fueron los primeros en presentar un modelo formal para regular el flujo de los datos dentro de un sistema de información. En su modelo cada sujeto y objeto dentro del sistema posee un determinado nivel de seguridad. La información puede ir desde cierto objeto a cierto sujeto sólo si el nivel de seguridad del sujeto es mayor al nivel de seguridad del objeto. En dicho modelo, también llamado *Multilevel Security* [BP73], dos propiedades de seguridad deben ser cumplidas: ningún proceso debe leer datos de un objeto con nivel mayor ("No Read Up") y ningún

proceso puede escribir datos en un objeto de nivel menor ("No Write Down"). Estas reglas eran impuestas por el sistema independientemente de las acciones de los usuarios. A diferencia de DAC, cuando es el sistema el que impone ciertas reglas se denomina *Mandatory Access Control*. De esta forma, un troyano que es ejecutado en cierto nivel, sólo puede operar dentro de ese nivel y no copiar datos confidenciales en objetos de menor nivel. Es posible aplicar estas políticas de seguridad para regular el flujo de la información asignando un nivel de seguridad a cada elemento del sistema y definiendo de qué manera el flujo de la información se puede propagar entre los diferentes niveles. Aplicando esta misma técnica al análisis del código móvil podemos controlar cómo los datos confidenciales del consumidor se propagan.

Sin embargo, en la práctica, este modelo resultó ser muy restrictivo. La performance se ve afectada por la necesidad de realizar los chequeos en tiempo de ejecución. Asimismo, se requiere almacenar información adicional en cada objeto y sujeto del sistema para reflejar el nivel asignado.

Pero lo que aún es más importante, es que dada su naturaleza dinámica para controlar el flujo de la información, es incapaz de detectar *flujos de información implícitos* [DD77]. Los flujos implícitos se generan por la estructura de control del programa a diferencia de los flujos explícitos causados por una asignación directa de datos confidenciales hacia datos públicos. Existen otros tipos de flujos de datos implícitos [VS97], por ejemplo aquellos que se generan a partir de la terminación o no terminación de la ejecución de un algoritmo (*termination channels*) o aquellos de los cuales podríamos deducir algo a partir de la hora en que una acción determinada ocurre (*timing channels*) [AGA00]. También sería posible obtener información a partir de la sobreutilización de un recurso (memoria o disco) compartido (*resource exhaustion channels*). Este trabajo se basa sólo en aquellos flujos implícitos que se generan a partir de la estructura sintáctica del programa.

Supongamos dos niveles de seguridad: **high** y **low**. Un dato de nivel **high** es considerado secreto o sensible mientras que un dato de nivel **low** es considerado público. El siguiente ejemplo presenta un flujo implícito desde una variable **high y** hacia una variable **low x**:

```
x := 0;
if (y = 0) then x := 1;
```

Este ejemplo es inseguro, debido a que la asignación $\mathbf{x := 1}$ sobre una variable **low** esta condicionada por el valor de la variable **high y** de la condición. Mandatory Access Control puede detectar esta asignación mediante una etiqueta o marca que indica el nivel de seguridad de los datos que un proceso esta manejando. La asignación $\mathbf{x := 1}$ es detectada en ejecución dado que un proceso de nivel **high** esta modificando un objeto de nivel **low**. Ahora, si consideramos el caso en donde $\mathbf{y \neq 0}$, ninguna asignación se lleva a cabo y por lo tanto ningún chequeo, lo cual nos permite deducir

que el valor de la variable **high y** es distinto de **0** observando que $\mathbf{x = 0}$. Determinar qué modificaciones son válidas requiere evaluar todos los posibles caminos que el flujo del programa puede tomar, lo cual es intratable a todo efecto práctico. Una forma de asegurar confidencialidad para este ejemplo, es que la etiqueta que indica el nivel de seguridad permanezca **high** todo el resto del programa, para que de esta forma todas las variables sean tratadas como confidenciales luego de que finalice la sentencia **if**. Esto lleva a realizar aumentos en el nivel de seguridad de la etiqueta haciendo que este modelo sea demasiado restrictivo para su uso.

1.5. Análisis de Flujo de Información Basado en Sistema de Tipos

Diferentes técnicas han sido estudiadas para dar solución al problema de regular el flujo de información. Comenzando con el trabajo de Bell y LaPadula [BP73] que proponen un mecanismo para regular el flujo de información en ejecución. Como se mencionó en la sección anterior, los mecanismos de control en ejecución, además de afectar la performance son incapaces de detectar flujos de información implícitos.

Denning y Denning [DD77] son los primeros en utilizar técnicas estáticas para controlar el flujo de información, eliminando la imposibilidad de detectar flujos implícitos. Su propuesta denominada *certificación de programas* presenta una forma de análisis del código en tiempo de compilación. Básicamente consiste en acciones a ejecutar por el compilador cuando una cadena perteneciente a cierta forma sintáctica es reconocida. Estas acciones son realizadas junto con el chequeo de tipos y la generación de código.

Una propuesta más reciente de realizar análisis estático de flujo de información es utilizando un *sistema de tipos* [VSI96]. Volpano, Smith e Irvine [VSI96], fueron los primeros en expresar la propiedad de no-interferencia en términos de expresiones de tipos. En su trabajo [VSI96] presentan un lenguaje de alto nivel simple, cuyo sistema de tipos expresa la propiedad de no-interferencia. También incluyen una prueba formal de que los programas escritos en su lenguaje que pasan el chequeo de tipos, cumplen con la propiedad de no-interferencia. Es decir, que pueden ser ejecutados con la seguridad que no contendrán flujos de información inválidos.

En un sistema de tipos con control de flujo de información, cada expresión tiene además de su tipo de dato, un nivel de seguridad el cual indica de qué manera ese dato puede ser utilizado. Por ejemplo, la siguiente declaración no solamente indica que **x** es una variable de tipo **int** sino que, además, indica que contendrá valores públicos.

```
int low: x;
```

Un programa se dice *seguro* si pasa el chequeo de tipos, asegurando que en ejecución no contendrá flujos inválidos. Al igual que el mecanismo presentado por Denning

y Denning [DD77], utilizando un sistema de tipos también es posible detectar flujos implícitos. Para poder detectarlos se utiliza una etiqueta de seguridad global denominada *program-counter* (**pc**) que captura la dependencia en cuanto al nivel de seguridad entre la condición y el cuerpo de una sentencia de control. Considerando el ejemplo presentado anteriormente, el valor del **pc**, luego de evaluada la condición, es **high**. La asignación **x := 1** es considerada segura si el nivel de seguridad de la variable **x** es como mínimo tan restrictiva como el nivel de seguridad del **pc**. Dado que el nivel de la variable **x** es **low**, el programa es rechazado por el chequeador de tipos como inseguro. La etiqueta **pc** se corresponde con la marca utilizada por el modelo dinámico, pero posee una diferencia importante, el nivel de seguridad del **pc**, luego de terminada la sentencia **if**, puede volver a ser el que tenía en la instrucción previa a dicha sentencia. Esto se puede realizar debido a que la instrucción inmediatamente posterior a la sentencia de control, se ejecutará independientemente del valor de la variable **y** en la condición. Como podemos observar, el análisis en tiempo de compilación puede mejorar la precisión. Esto se debe a que el modelo dinámico sólo posee información de un único camino de ejecución, mientras que el modelo estático puede asegurar que todos los posibles caminos de ejecución de una sentencia **if** no poseen asignaciones inválidas.

Este mecanismo es el que cuenta con mayor aceptación para asegurar el flujo de información, aunque también posee algunas limitaciones. Por ejemplo, los siguientes programas serán rechazados como inseguros por un sistema de tipos como el desarrollado en [VSI96]. Mientras que ambos cumplen con la propiedad de no-interferencia.

```
(a) int low: x; int high: y;
    y := 1;
    if (y = 1) then x := 1 else x := 2;


(b) int low: x; int high: y;
    if (y = 1) then x := 1 else x := 2; x := 1;
```

1.6. No-Interferencia

De forma de tratar rigurosamente qué queremos decir con flujo de información seguro y poder probar que el análisis realizado es correcto, el concepto de seguridad en el flujo de información es desarrollado en función de lo que es conocido como *no-interferencia* (en inglés, "noninterference") [GM82]. No-interferencia formaliza el hecho de que el uso de datos sensibles no interfiere o afecta datos públicos de la siguiente manera: si tomamos dos ejecuciones diferentes de un *mismo* programa cuyos valores de entrada difieren, a lo sumo, en las variables declaradas como **high**, los valores de salida **low** no deben verse afectados al final de las ejecuciones. El siguiente programa por ejemplo,

```
int low: x; int high: y;
y := x + 1;
```

es seguro dado que los valores de la variable **low x** no se ven afectados, por cambios hechos sobre la variable **high y**. Sin embargo, el programa,

```
int low: x; int high: y;
x := y;
```

es inseguro dado que si tomamos 2 y 3 como los valores iniciales de la variable **high y**, los valores de salida de la variable **low x** varían.

1.7. Lenguajes de Bajo Nivel

Tal como se describe en [SM03] mucho esfuerzo ha sido focalizado en lenguajes de alto nivel, pero poco en asegurar no-interferencia en lenguajes de bajo nivel. Controlar el flujo de información en lenguajes lo más cercano posible al código que ejecutará la máquina tiene sus ventajas por sobre los lenguajes de alto nivel. En primer lugar, mucho código malintencionado es distribuido luego de ser compilado y segundo, si el análisis de flujo de información ha sido realizado sobre el código fuente, tenemos que confiar que el código compilado preserva las mismas propiedades de seguridad. Por otro lado, realizar el análisis de flujo de información en los lenguajes de bajo nivel tiene una complicación adicional debido a que los programas carecen de estructura sintáctica. Asimismo, y a favor de la flexibilidad en la práctica, es importante contar con una función de compilación que traduzca programas de alto nivel escritos en un lenguaje con análisis de flujo de información, a un lenguaje de bajo nivel con anotaciones que permitan la validación de que el flujo es seguro.

Otra complicación que presentan los lenguajes tipo assembler por sobre los lenguajes de alto nivel, es la utilización que estos realizan de la pila de ejecución. Dado que la técnica de compilación más difundida se basa en el uso de una pila de ejecución, es necesario incorporar el tratamiento y manipulación de la pila de forma tal que garantice que no se filtrará información a través de su utilización. Por ejemplo, aquellos elementos que sean agregados a la pila con cierto nivel de seguridad l, deben ser recuperados con, como mínimo, con el mismo nivel de seguridad l. En el siguiente capítulo se describen otras formas más sutiles de filtrado de información a través del uso de la pila.

1.8. Trabajos Relacionados

Seguidamente presentamos algunos trabajos destacados en el área con la finalidad de encuadrar el contexto del presente trabajo. De ninguna manera se pretende dar una lista completa.

1.8.1. Lenguajes de Alto Nivel

El primer trabajo en establecer formalmente que un programa correctamente tipado verifica no-interferencia se presenta en [VSI96]. Allí se describe un sistema de tipos con niveles de seguridad para un lenguaje con procedimientos de primer orden y se prueba que garantiza no-interferencia. Luego, el foco en investigación sobre el análisis de flujo de información se centró en aumentar las características de los lenguajes de programación. Banerjee y Nauman [BN02] desarrollaron un sistema de tipos extendiendo el presentado en [VSI96] para un lenguaje orientado a objetos basado en Java y prueban que garantiza no-interferencia. Este lenguaje incluye, entre otras, las siguientes características: punteros, accesibilidad de las variables (públicas y privadas), binding dinámico y herencia. Otra construcción importante que se estudió fue el manejo de excepciones. El hecho de que una excepción pueda propagarse daba lugar a la generación de flujos implícitos. Volpano y Smith [VSI96] proponen un sistema de tipos para controlar los flujos implícitos causados por la propagación de excepciones que mas tarde es mejorado por Myers [MYE99]. El análisis de flujo de información en lenguajes concurrentes también es objeto de estudio. Volpano y Smith [SV98] probaron no-interferencia para un lenguaje multithreaded. Sabelfeld [SAB01] extendió el sistema de tipos para manejar sincronización entre threads. Recientemente Alejandro Russo en su tesis de doctorado [RUS08] propone nuevas técnicas para evitar filtrado de información producido en lenguajes multithreaded cuando variables privadas afectan el tiempo de ejecución de un thread. Cuando esto sucede, el scheduler puede pasar el control a otro thread afectando el orden de cómo las variables públicas son modificadas. Esto se conoce como *Internal Timing Covert Channel* [RUS08].

Heintze y Riecke [HR98] trabajaron en el análisis de flujo de información en un lenguaje del paradigma funcional denominado SLam Calculus. En su trabajo presentan un sistema de tipos con niveles de seguridad para SLam y la prueba de que asegura no-interferencia. Luego, Pottier y Simonet [PS02] extendieron el sistema de tipos de SLam con referencias, manejo de excepciones y polimorfismo y probaron no-interferencia.

1.8.2. Implementaciones

En la actualidad existen pocas implementaciones de lenguajes que realicen análisis de flujo de información. Esto se debe, en parte, a las limitaciones y restricciones de varios de los modelos existentes.

La primera implementación de envergadura fue JFlow [MYE99] una extensión del lenguaje Java [GJS96] que permite el análisis estático de flujo de información. Una importante característica de JFlow por sobre otros trabajos en chequeo estático de flujo de información, es que fue pensado para un modelo de programación que sea utilizable en la práctica. Dado que el autor no apunta a contar con una demostración formal de la correctitud de su sistema (tal demostración se encuentra inexistente al

día de la fecha), esto permite una flexibilidad considerable para escribir programas de forma natural. Algunas de la características que favorecen esta flexibilidad son:

- *Label polymorphism* consiste en asignar *variables de seguridad* a las variables del programa (en lugar de constantes como *low* y *high*). Luego el sistema determina las restricciones que deben imponerse a las variables de seguridad para que el programa resultante sea seguro.
- La posibilidad de insertar chequeos en tiempo de ejecución en el caso de que el sistema no pueda determinar que el flujo de información no es seguro.
- Inferencia automática, siempre que sea posible, de los niveles de seguridad de los datos.

El compilador de JFlow denominado JIF [MNZZ99], recibe un programa al cual le aplica el chequeo de tipos, luego elimina las anotaciones y lo traduce a código Java que puede ser compilado por cualquier compilador de Java.

Flow Caml [SIM03] es una extensión del lenguaje Objective Caml con un sistema de tipos con niveles de seguridad. Al igual que JIF, Flow Caml toma como entrada un programa fuente, realiza el chequeo del flujo de información respecto de las políticas de seguridad especificadas por el programador y produce código Objective Caml que puede ser compilado por cualquier compilador de este lenguaje.

1.8.3. Lenguajes de Bajo Nivel

Entre los trabajos que estudiaron el análisis de flujo de información en lenguajes de bajo nivel se encuentran [YI06], [BCM04] y [BBR04]. En [BBR04] se presenta un sistema de tipos con niveles de seguridad para un lenguaje de bytecode. También presentan un lenguaje de alto nivel tipo *while* y su traducción con preservación de tipos. El lenguaje de bytecode presentado tiene importantes limitaciones detalladas en [BB08]. En [BB08] se mejora la precisión del análisis del flujo de información para bytecode, sin relacionar los resultados con un lenguaje de alto nivel. En [YI06] se presenta un lenguaje tipo RISC, llamado TAL_c, con un sistema de tipos con niveles de seguridad. Los autores prueban que los programas bien tipados garantizan no-interferencia. También incluyen una función de compilación con preservación de tipos. En [BCM04] se presenta SIFTAL un lenguaje basado en TAL [WAL99] con un sistema de tipos con niveles de seguridad. Los autores demuestran que los programas bien tipados garantizan no-interferencia. Existen diferencias en cuanto a la forma en que los autores de [YI06] y [BCM04] solucionaron el problema que presentan los lenguajes de bajo nivel para detectar flujos de información implícitos. Si bien el trabajo presentado en [YI06] posee una función de compilación que no se encuentra presente en [BCM04] [MED06], TAL_c posee la limitación de requerir pilas con la misma longitud durante la ejecución de

las ramas **high** de una sentencia condicional. Esto reduce la cantidad de programas que cumplen con no-interferencia que serán aceptados por el sistema de tipos. Otras diferencias y con mayor detalle entre estos dos trabajos en [MED06].

1.9. Contribuciones y Estructura del Trabajo

Este trabajo primero presenta un lenguaje de alto nivel junto con un sistema de tipos que garantiza que los programas bien tipados satisfacen no-interferencia. Luego presenta un lenguaje de bajo nivel basado en Typed Assembly Language [WAL99], al que denominamos SecTAL (Secure Typed Assembly Language), junto con un sistema de tipos que satisface propiedades similares. Finalmente, presenta una función de compilación junto con una demostración de que preserva la propiedad de no-interferencia. La función de compilación ha sido implementada al igual que un chequeador de tipos para SecTAL[1].

Las contribuciones pueden resumirse de la siguiente manera:

- La definición de una función de compilación de un lenguaje imperativo sencillo hacia un lenguaje de bajo nivel basado en Typed Assembly Language.
- La prueba de un resultado de preservación de tipado que muestra que si el programa fuente es bien tipado (y por ende seguro) también lo será el resultado de compilar el mismo.

Un resumen de este trabajo fue publicado en ASSE'09 [EBM09].

Estructura del trabajo. Los dos siguientes capítulos presentan el lenguaje objeto y el lenguaje fuente de la función de compilación. El capítulo 2 presenta el lenguaje objeto. Este capítulo tiene el propósito de presentar con más detalle el lenguaje objeto de modo que sirva de introducción al capítulo 3 donde se presenta la función de compilación y la prueba de preservación de tipos. En el capítulo 4 se describe la implementación realizada y las herramientas utilizadas para el desarrollo del compilador. El capitulo 5 presenta las conclusiones finales. El apéndice A muestra las demostraciones de no-interferencia del lenguaje de alto nivel.

[1] Para más detalles consultar `www.lifia.info.unlp.edu.ar/~eduardo/compilacionSegura/index.html`.

Capítulo 2

El Lenguaje Objeto

En este capítulo describiremos SecTAL (Secure Typed Assembly Language), el lenguaje objeto de nuestra función de compilación. SecTAL es una variante del lenguaje SIFTAL [BCM04, MED06] que permite reuso de registros. Ambos están equipados con un sistema de tipos que garantiza no-interferencia, con la diferencia que SIFTAL requiere que los registros mantengan su nivel de seguridad a lo largo de todo el proceso de validación del flujo de información. Pero esto no es aceptable [MED06] dado que la intención es validar el flujo de información sobre código generado por compiladores que utilizan prácticas estándares reutilizando libremente los registros. SIFTAL al igual que SecTAL son lenguajes que pertenecen a la familia de lenguajes assembler tipados como TAL [WAL99] y STAL [MCGW02].

2.1. Complicaciones en Lenguajes de Bajo Nivel

2.1.1. Falta de Estructura Sintáctica

Como se mencionó anteriormente una de las mayores complicaciones que poseen los lenguajes de bajo nivel es que carecen de estructura sintáctica. Esto impide reconocer fácilmente dónde termina una bifurcación condicional (tanto sentencias condicionales como iterativas) para que sea seguro bajar el nivel de seguridad del **pc**. Si bien nuestro lenguaje objeto es SecTAL, y haremos referencia a él, cabe destacar que esto ha sido resuelto por los autores de SIFTAL [BCM04, MED06], y adoptado en SecTAL sin cambios. Este problema es resuelto utilizando lo que se denomina *junction point*, aquellos puntos de un programa en donde las ramas de una bifurcación condicional convergen. Dado que las bifurcaciones condicionales pueden estar anidadas, el sistema de tipos de SecTAL maneja una estructura de pila de *junction points* para asegurar que los programas estén correctamente estructurados. Para poder manipular esta pila se utilizan las instrucciones `jumpJP` y `pushJP`, que proveen al sistema de tipos información extra para realizar el análisis correspondiente. Estas instrucciones en realidad

```
pc level      x : int^⊤
              y : int^⊤;
              z : int^⊥
  low         if x = 0
  high          then y := 1
  high          else y := 2;
  low         z := 3
```

(a) Language de alto nivel

```
L1 :  bnz r1, L2
      mov r2, 1
      jmp L3
L2 :  mov r2, 2
L3 :  mov r3, 3
```

(b) Lenguaje assembler

```
L1 :  pushJP L3
      bnz r1,L2
      mov r2, 1
      jmpJP L3
L2 :  mov r2, 2
      jmpJP L3
L3 :  mov r3, 3
```

(c) SecTAL

Figura 2.1: Ejemplo de un flujo de información implícito.

son directivas de tipado. Una vez terminado el chequeo de tipos, la instrucción `pushJP` puede ser eliminada y la instrucción `jumpJP` puede ser reemplazada por una instrucción `jmp` convencional, para que luego el programa pueda ser ejecutado por una máquina. El ejemplo de la figura 2.1 describe la utilización de estas dos instrucciones.

La figura 2.1 (a) presenta un programa en un lenguaje de alto nivel que cumple con la propiedad de no-interferencia. El lenguaje de alto nivel utilizado para el ejemplo posee solo dos niveles de seguridad: *low* y *high*. Para ilustrar el uso de las directivas de tipado, el programa de alto nivel declara tres variables: x de tipo *entero high* ($\texttt{int}^{\top}$), y también de tipo *entero high* y z de tipo *entero low* ($\texttt{int}^{\bot}$). Allí se puede apreciar que la asignación $z := 3$ no depende de la variable *entera high* x con lo cual es seguro bajar el **pc** a *low*. El programa de la figura 2.1 (b) es la traducción estándar al lenguaje assembler. Note aquí que el **pc** puede ser aumentado a *high* después de la ejecución de la instrucción `bnz`, pero no hay forma de determinar cuando puede ser bajado a *low* para poder realizar la modificación de la variable pública z, representada por el registro r_3. El programa de la figura 2.1 (c) es la traducción a SecTAL (sin la declaración de los bloques de código para simplificar) del programa de la figura 2.1 (a). Este ejemplo en SecTAL utiliza la etiqueta *L3* como *junction point*, para determinar el punto donde la implícita instrucción *if* converge y así poder bajar el **pc** a *low*. La instrucción `pushJP` inserta en el tope de la pila la etiqueta *L3* que solo puede ser removida por la instrucción `jumpJP`. De esta forma, el bloque de código etiquetado por *L3* puede iniciar su ejecución con un **pc** público.

2.1.2. Estructura de Pila

La sección anterior explica la dificultad (y su solución) que se presenta en los lenguajes de bajo nivel para determinar la estructura sintáctica de un programa. En esta sección abordamos las complicaciones adicionales que trae la incorporación de una pila de control. El uso de la pila permite diferentes y nuevas formas de generar flujos de información inválidos. Al igual que en la sección anterior el sistema de tipos de SIFTAL [BCM04, MED06] controla el uso de la pila, garantizando que no habrá flujos de

información inválidos. Esto fue adoptado en SecTAL sin cambios.

Con la intención de que los ejemplos desarrollados a continuación sean claros, explicamos brevemente parte de la sintaxis de SecTAL. La siguiente sección se encarga de definir la sintaxis con precisión.

SecTAL posee instrucciones para manipular la pila de control, a saber: `salloc` para asignar espacio en el tope, `sfree` para liberar espacio, `sld` para leer valores de la pila y `sst` para almacenar valores en la pila. Considere el ejemplo de la figura 2.2. La expresión $\text{Code}\langle\Gamma \mid \texttt{s} \mid \textbf{pc}\rangle$ junto al nombre de la etiqueta es el tipo del bloque de código, donde Γ contiene los registros y sus tipos, antes del comienzo de la ejecución. `s` es la pila de *junction points* explicada en la sección anterior y **pc** indica el nivel de seguridad inicial. En Γ contamos con el registro de puntero de pila `sp`, el cual apunta al inicio de la pila. Por ejemplo, si tenemos $\texttt{sp} : int^{\top}.int^{\bot}.\epsilon$, decimos que la pila contiene dos elementos, el primero es un *entero high* y el segundo es un *entero low*. La figura 2.2 muestra un ejemplo de flujo de información inválido que se genera por el uso de la pila de control. El programa comienza en el bloque de código L_{start}, con el tope de la pila con tipo *entero low*.

L_{start} $\quad \text{Code}\langle\{\texttt{r0} : int^{\bot}, \texttt{r1} : int^{\bot}, \texttt{r2} : int^{\top}, \texttt{sp} : int^{\bot}.\epsilon\} \mid \epsilon \mid \bot\rangle$

```
mov r0, 0
mov r1, 1
pushJP L_JP
bnz r2, L_high
sst sp[0], r0
jmpJP L_JP
```

L_{high} $\quad \text{Code}\langle\{\texttt{r0} : int^{\bot}, \texttt{r1} : int^{\bot}, \texttt{r2} : int^{\top}, \texttt{sp} : int^{\bot}.\epsilon\} \mid L_{JP}.\epsilon \mid \top\rangle$

```
sst sp[0], r1
jmpJP L_JP
```

L_{JP} $\quad \text{Code}\langle\{\texttt{r0} : int^{\bot}, \texttt{r1} : int^{\bot}, \texttt{r2} : int^{\top}, \texttt{sp} : int^{\bot}.\epsilon\} \mid \epsilon \mid \bot\rangle$

```
sld r1, sp[0]
...
```

Figura 2.2: Flujo de información inválido generado a través del uso de la pila

Luego de la inicialización de los registros `r0` y `r1`, el programa prepara la pila de *junction points* almacenando la etiqueta L_{JP} y ejecuta el salto condicional que depende del valor *high* almacenado en el registro `r2`. Note que en las dos ramas de la bifurcación condicional se cambia el valor almacenado en el tope de la pila, mediante el uso de la instrucción `sst`. En una de las ramas el valor es 0 y en la otra es 1. Dado que los valores son diferentes, cuando el programa llega al *junction point* L_{JP}, se puede generar un flujo de información inválido por el uso de la instrucción `sld` que recupera el tope de la

pila y lo guarda en el registro *low*, `r1`. Es decir, para diferentes valores de entrada en el registro *high* `r2`, el registro *low* `r1` puede terminar con valores diferentes. El sistema de tipos de SecTAL rechazará este tipo de programas al considerarlos inválidos por no permitir almacenar en la pila un valor de menor nivel de seguridad que el **pc** actual (ver la regla de tipado T-Sst).

Otras formas de flujos de información inválidos se generan alterando el tamaño de la pila cuando el **pc** es *high*. El programa de la figura 2.3 asigna espacio en la pila para almacenar dos valores *enteros low*, 0 y 1. Luego prepara la pila de *junction points* almacenando la etiqueta L_{JP}, y ejecuta el salto condicional que depende del valor *high* almacenado en el registro `r2`. La rama cuya etiqueta es L_{high}, libera el tope de la pila utilizando la instrucción `sfree`. Cuando el programa llega al *junction point* L_{JP}, se puede generar un flujo inválido, dado que el valor que se asignará al registro `r1` dependerá de la rama ejecutada, es decir, del valor almacenado en el registro *high* `r2`. Podríamos evitar este flujo de información inválido si prohibiéramos liberar espacios *low* de la pila cuando el **pc** es *high*. Sin embargo, esto no es suficiente.

El ejemplo de la figura 2.4 presenta un caso similar, pero el valor liberado de la pila es *high* en lugar de *low*. El valor *high* del tope de la pila es liberado únicamente cuando la rama cuya etiqueta es L_{high} es ejecutada. Cuando el programa llega al *junction point* L_{JP}, se puede generar un flujo inválido dado que el valor que se asignará al registro `r1` dependerá de la rama ejecutada, es decir del valor almacenado en el registro *high* `r2`. Si la rama ejecutada es L_{high} el valor final en `r1` será 1 y sino será 0.

Por los ejemplos de las figuras 2.3 y 2.4 se puede concluir que la causa de la generación de flujos de información inválidos no es el nivel de seguridad que posee el elemento a liberar de la pila sino la variación del tamaño de la pila. La propuesta de los autores de SIFTAL y adoptada en SecTAL para evitar estos flujos inválidos, es combinar la pila de *junction points* con la pila de control. De esta forma los *junction points*, además de la función explicada en la sección anterior, se utilizan como *marcas* dentro de la pila de control. Estas *marcas* indican que partes de la pila de control pueden asignarse y modificarse durante la ejecución de una rama con **pc** high (aquellas por encima de la etiqueta del *junction point*), y cuales deben mantenerse sin alteraciones (aquellas por debajo de la etiqueta del *junction point*).

Las etiquetas agregadas como tipos en la pila de control, sólo serán utilizadas durante el chequeo de tipos y nunca durante la ejecución. Se puede observar en la figura 2.7, que la regla SO_Push no produce ningún efecto sobre la pila en ejecución.

2.1.3. Tipos Polimórficos

Como habrán notado, en los ejemplos de las figuras 2.3 y 2.4 se utilizan (...), en los tipos de la pila (`sp`), en el bloque de código L_{JP}, para ocultar la necesidad

L_{start} $\text{CODE}\langle\{\texttt{r1} : int^{\perp}, \texttt{r2} : int^{\top}, \texttt{sp} : \epsilon\} \mid \epsilon \mid \perp\rangle$
`salloc` 2
`mov` $r1, 0$
`sst` `sp`$[0], r1$
`mov` $r1, 1$
`sst` `sp`$[1], r1$
`pushJP` L_{JP}
`bnz` $r2, L_{high}$
`jmpJP` L_{JP}

L_{high} $\text{CODE}\langle\{\texttt{r1} : int^{\perp}, \texttt{r2} : int^{\top}, \texttt{sp} : int^{\perp}.int^{\perp}.\epsilon\} \mid L_{JP}.\epsilon \mid \top\rangle$
`sfree` 1
`jmpJP` L_{JP}

L_{JP} $\text{CODE}\langle\{\texttt{r1} : int^{\perp}, \texttt{r2} : int^{\top}, \texttt{sp} : int^{\perp}....\} \mid \epsilon \mid \perp\rangle$
`sld` $r1,$ `sp`$[0]$
...

Figura 2.3: Flujo de información inválido generado por quitar de la pila una posición con nivel de seguridad low

de requerir tipos *polimórficos* que nos permitan tipar dichos programas. Tomando el ejemplo de la figura 2.3, se puede ver que los tipos de la pila son $\texttt{sp} : int^{\perp}....$ para el bloque de código L_{JP}. Mientras que los tipos de la pila para una de las ramas de la bifurcación es $\texttt{sp} : int^{\perp}.int^{\perp}.\epsilon$ y para la otra (que quita de la pila un elemento) es $\texttt{sp} : int^{\perp}.\epsilon$. Podemos notar que las pilas coinciden en el tipo del tope y difieren en el resto. Entonces, en la definición del bloque de código que representa el *junction point* necesitamos expresar el tipo esperado en el tope y dejar que el resto sea cualquier tipo ($int^{\perp}.\epsilon$ y ϵ respectivamente). En general, queremos expresar el tipo de ciertos elementos del tope y dejar que el resto sea *instanciado* con tipos diferentes. Para esto utilizamos tipos variable de pila [MCGW02]. Este mecanismo es necesario para la implementación de funciones recursivas, ya que en estos casos cada nueva llamada recursiva requiere almacenar en la pila la dirección de retorno, con lo cual se obtendría una pila diferente.

Si utilizamos el tipo variable de pila X, entonces la definición de la pila para el bloque de código L_{JP} de la figura 2.3, se podría volver a escribir de la siguiente forma: $int^{\perp}.X$. Luego, para *instanciar* dicha variable, deberíamos cambiar el primer `jmpJP` a `jmpJP` $L_{JP}[int^{\perp}.\epsilon]$ y el segundo a `jmpJP` $L_{JP}[\epsilon]$.

2.2. Sintaxis

Un programa SECTAL es un conjunto de bloques etiquetados de instrucciones. A cada bloque se le asigna un tipo que indica los requisitos que deben cumplir los

L_{start} $\text{CODE}\langle\{\texttt{r1} : int^{\bot}, \texttt{r2} : int^{\top}, \texttt{sp} : int^{\top}.int^{\bot}.int^{\bot}.\epsilon\} \mid \epsilon \mid \bot\rangle$
`mov` $r1, 0$
`sst sp`$[1], r1$
`mov` $r1, 1$
`sst sp`$[2], r1$
`pushJP` L_{JP}
`bnz` $r2, L_{high}$
`jmpJP` L_{JP}

L_{high} $\text{CODE}\langle\{\texttt{r1} : int^{\bot}, \texttt{r2} : int^{\top}, \texttt{sp} : int^{\top}.int^{\bot}.int^{\bot}.\epsilon\} \mid L_{JP}.\epsilon \mid \top\rangle$
`sfree` 1
`jmpJP` L_{JP}

L_{JP} $\text{CODE}\langle\{\texttt{r1} : int^{\bot}, \texttt{r2} : int^{\top}, \texttt{sp} : int^{\top}.int^{\bot}.\ldots.\} \mid \epsilon \mid \bot\rangle$
`sld` $r1$, `sp`$[1]$
$\ldots$

Figura 2.4: Flujo de información inválido generado por quitar de la pila una posición con nivel de seguridad high

registros y la pila de control para que ese bloque pueda ser ejecutado. La figura 2.5 muestra la sintaxis de los tipos. En SecTAL se define un orden parcial para los niveles de seguridad, denotado por $\mathfrak{L}_{\mathbf{sec}}$. Los elementos menor y mayor de dicho orden son $\bot$ y $\top$, respectivamente. Utilizamos $\sqcup$ (*junta*) para obtener el supremo entre dos elementos, y $\sqsubseteq$, para indicar el orden. Nótese, que si bien SecTAL es un lenguaje rico en niveles de seguridad, éste será utilizado como lenguaje objeto del lenguaje de alto nivel que se define en el capítulo 3, al cual denominamos While. Como veremos más adelante, While posee dos niveles de seguridad, L (low) y H (high). Con lo cual, sólo utilizaremos los elementos menor y mayor de $\mathfrak{L}_{\mathbf{sec}}$, $\bot$ y $\top$ para la traducción.

Enteros (*int*), tuplas ($\langle\sigma_0, \ldots, \sigma_n\rangle$ o $\langle\overline{\sigma}\rangle$) y los tipos de los bloques de código ($\texttt{CODE}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle$) son *tipos básicos* ($\tau$). Estos *tipos básicos* son anotados con un nivel de seguridad l que se obtiene de *niveles de seguridad* para formar los *tipos con nivel de seguridad* (σ). Los tipos de los bloques de código se forman por el contexto de *tipado de registros*, Γ, que mapea registros a sus tipos (incluye el registro `sp`), el program-counter **pc** y una lista de tipos variables de pila Θ. Θ liga las variables libres en Γ. Escribiremos X, Θ para denotar una secuencia de tipos variables de pila, donde X esta en el primer lugar y no esta incluida en Θ. Los *tipos de la pila de control* (Σ) son secuencias de tipos de componentes de la pila y etiquetas de bloques de código para referirnos a *junction points*. El tipo *nonsense* es utilizado cuando se reserva espacio en la pila. Indica posiciones de pila no inicializadas. Usamos $FV(\Sigma)$ para obtener las variables libres en Σ. El *tipado del heap* mapea direcciones de memoria del heap a tipos

niveles de seguridad	$l, \mathbf{pc}$	$\in$	$\mathfrak{L}_{\mathbf{sec}}$
tipos con nivel de seguridad	σ	$::=$	τ^l
tipos básicos	τ	$::=$	$int \mid \langle \sigma_0, \ldots, \sigma_n \rangle \mid \texttt{CODE}\langle \forall[\Theta]\Gamma \mid \mathbf{pc} \rangle$
tipado de registros	Γ	$::=$	$\{r_1 : \sigma_1, \ldots, r_n : \sigma_n, \texttt{sp} : \Sigma\}$
tipos de la pila de control	Σ	$::=$	$X \mid \epsilon \mid \hat{\sigma} \cdot \Sigma \mid L \cdot \Sigma$
tipos de componentes de la pila de control	$\hat{\sigma}$	$::=$	$\sigma \mid ns$
tipado del heap	Ψ	$::=$	$\{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n\}$
tipado de la configuración de máquina SecTAL	Ω	$::=$	$[\Psi, \Gamma, \mathbf{pc}]$

Figura 2.5: Sintaxis de tipos de SecTAL

configuración de máquina SecTAL	Π	$::=$	(H, R, B)
heap	H	$::=$	$\{\ell_1 : h_1, \ldots, \ell_n : h_n\}$
etiquetas del heap	ℓ	$::=$	$p \mid L$
valores del heap	h	$::=$	$\langle w, \ldots, w \rangle \mid \texttt{CODE}\langle \forall[\Theta]\Gamma \mid \mathbf{pc} \rangle^l.B$
registros	R	$::=$	$\{r_1 : w_1, \ldots, r_n : w_n, \texttt{sp} : S\}$
bloques de código	B	$::=$	$\texttt{halt} \mid \texttt{jmp}\ v \mid \texttt{jmpJP}\ L[\Sigma] \mid \iota; B$
instrucciones	ι	$::=$	$\texttt{aop}\ r_d, r_s, v \mid \texttt{bnz}\ r, v \mid \texttt{mov}\ r, v \mid \texttt{ld}\ r_d, r_s[i]$
			$\texttt{st}\ r_d[i], r_s \mid \texttt{pushJP}\ L \mid \texttt{salloc}\ i \mid \texttt{sfree}\ i$
			$\texttt{sld}\ r_d, \texttt{sp}[i] \mid \texttt{sst}\ \texttt{sp}[i], r_s \mid \texttt{sstNsl}\ level, \texttt{sp}[i], r_s$
operaciones aritméticas	$\texttt{aop}$	$::=$	$\texttt{add} \mid \texttt{sub} \mid \texttt{mul}$
operandos	v	$::=$	$r \mid w \mid v[\Sigma]$
valores	w	$::=$	$i \mid p \mid L \mid w[\Sigma]$
valores de componentes de la pila	$\hat{w}$	$::=$	$w \mid ns$
pila	S	$::=$	$\epsilon \mid \hat{w} \cdot S$

Figura 2.6: Sintaxis de SecTAL

con niveles de seguridad. Ψ mapea tipos de bloques de código a sus etiquetas L y tipos de tuplas a etiquetas de tupla p. Finalmente, el tipado de la configuración de máquina Ω requiere información de tipado de Ψ, Γ y $\mathbf{pc}$.

La figura 2.6 muestra la sintaxis de la configuración de máquina. Una *configuración de máquina* SecTAL es una tupla (H, R, B) donde H es el *heap*, mapea *etiquetas* a *valores del heap*, R representa el *conjunto de registros*, mapea registros a *valores* y B es el *bloque de código* que se esta ejecutando. ℓ es un bloque de código L o una etiqueta de tupla[1] p. Un valor h almacenado en el heap es un bloque de código con nivel de seguridad o una tupla de valores. Los valores pueden ser constantes enteras (i), una etiqueta p o una etiqueta seguida de una serie de tipos de la pila de la forma $L[\Sigma_1] \ldots [\Sigma_n]$. El sistema de tipos no permite tipar cadenas con la forma $i[\Sigma] \ldots [\Sigma]$ ni

[1] Tener en cuenta que para la función de compilación no se utilizaron tuplas.

$p[\Sigma]\ldots[\Sigma]$. El conjunto de registros r_i, incluye `sp` que apunta al inicio de la pila de control. Un operando v puede ser un *registro* o un *valor*. Para obtener el tipo de un operando, utilizamos la función $\Psi \cup \Gamma$, definida a continuación:

$$(\Psi \cup \Gamma)(v) \stackrel{\text{def}}{=} \begin{cases} \Gamma(r) & \text{si } v = r \\ \Psi(v) & \text{si } v = p \text{ or } v = L \\ int^{\perp} & \text{si } v = i \\ \texttt{CODE}\langle \forall[\Theta]\Gamma^{\Sigma}_{X} \mid \mathbf{pc}\rangle^{l} & \text{si } v = v'[\Sigma] \text{ y} \\ & (\Psi \cup \Gamma)(v') = \texttt{CODE}\langle \forall[X,\Theta]\Gamma \mid \mathbf{pc}\rangle^{l} \end{cases}$$

Las instrucciones de SecTAL están basadas en las instrucciones de STAL [MCGW02]. Como se mencionó anteriormente, en SIFTAL y adoptado en SecTAL, además de las instrucciones assembler estándar se incluyen dos directivas de tipado `jumpJP` y `pushJP` para manipular la pila de junction points. SecTAL además incluye otra directiva de tipado `sstNsl` la cual puede reemplazarse por la instrucción `sst` después de realizado el chequeo de tipos. De igual forma que en SIFTAL la directiva de tipado `jumpJP` se reemplaza por la instrucción `jmp`. `sstNsl` $level, \texttt{sp}[i], r$ guarda en la pila el registro r con tipo $level$ independientemente del tipo actual de r. En la sección 3.2.2 se explica con más detalle el uso de dicha directiva de tipado.
Como se mencionó anteriormente, las instrucciones de salto (`jmp`, `bnz`, `jmpJP`) pueden instanciar los tipos variables de pila del bloque de código de destino. Para implementar dicha instanciación, definimos la noción de *sustitución* de tipos variables de pila por tipos de pila.
Una sustitución γ es un conjunto finito de pares (X, Σ), con X siendo el tipo variable de pila y Σ el tipo de pila. Usamos Γ^{Σ}_{X} para indicar la sustitución de todas las ocurrencias libres del tipo variable de pila X en Γ con Σ. Esta notación también es utilizada para indicar sustitución en todas las categorías sintácticas, valores, operandos, tipos, bloques de código, etc. Una aplicación de una sustitución en Γ se escribe así $\Gamma\{\gamma\}$.

Si Θ es una secuencia de tipos variable de pila $X_1, \ldots, X_n$ y $\overline{\Sigma}$ es una secuencia de tipos de pila $\Sigma_1, \ldots, \Sigma_n$, entonces escribimos $\Gamma^{\overline{\Sigma}}_{\Theta}$ por $\Gamma^{\Sigma_1\,\Sigma_2\,\cdots\,\Sigma_n}_{X_1 X_2 \cdots X_n}$, si $\{X_i, \ldots, X_n\} \cap FV(\Sigma_{i-1}) = \emptyset$, para $2 \leq i \leq n$.

La definición precisa para cada una de las categorías sintácticas se muestra a continuación.

- Valores

$$\begin{aligned} i_X^\Sigma &\stackrel{\text{def}}{=} i \\ L_X^\Sigma &\stackrel{\text{def}}{=} L \\ p_X^\Sigma &\stackrel{\text{def}}{=} p \\ w[\Sigma_1]_X^\Sigma &\stackrel{\text{def}}{=} w_X^\Sigma[{\Sigma_1}_X^\Sigma] \end{aligned}$$

- Operandos

$$\begin{aligned} r_X^\Sigma &\stackrel{\text{def}}{=} r \\ v[\Sigma_1]_X^\Sigma &\stackrel{\text{def}}{=} v_X^\Sigma[{\Sigma_1}_X^\Sigma] \end{aligned}$$

- Pila

$$\begin{aligned} ns_X^\Sigma &\stackrel{\text{def}}{=} ns \\ \epsilon_X^\Sigma &\stackrel{\text{def}}{=} \epsilon \\ \hat{w} \cdot S_X^\Sigma &\stackrel{\text{def}}{=} \hat{w}_X^\Sigma \cdot S_X^\Sigma \end{aligned}$$

- Tipos

$$\begin{aligned} int_X^\Sigma &\stackrel{\text{def}}{=} int \\ \texttt{CODE}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle_X^\Sigma &\stackrel{\text{def}}{=} \texttt{CODE}\langle\forall[\Theta']\Gamma\{\gamma\}_X^\Sigma \mid \mathbf{pc}\rangle \\ &\quad \text{con } \gamma \text{ renombre } \Theta \text{ hacia } \Theta' \\ &\quad \text{y } \Theta' \text{ fresh } (\Theta' \not\subseteq FV(\Sigma) \cup \{X\}) \\ \langle\sigma_1, \ldots, \sigma_n\rangle_X^\Sigma &\stackrel{\text{def}}{=} \langle {\sigma_1}_X^\Sigma, \ldots, {\sigma_n}_X^\Sigma\rangle \end{aligned}$$

- Tipos con nivel de seguridad

$$\tau^{l\,\Sigma}_X \stackrel{\text{def}}{=} \tau_X^{\Sigma\,l}$$

- Tipado de registros

$$\begin{aligned} &\{r_1 : \sigma_1, \ldots, r_n : \sigma_n, \texttt{sp} : \Sigma_1\}_X^\Sigma \\ &\stackrel{\text{def}}{=} \{r_1 : {\sigma_1}_X^\Sigma, \ldots, r_n : {\sigma_n}_X^\Sigma, \texttt{sp} : {\Sigma_1}_X^\Sigma\} \end{aligned}$$

- Tipos de la pila de control y tipos de componentes de la pila

$$
\begin{array}{rcl}
ns^{\Sigma}_{X} & \stackrel{\text{def}}{=} & ns \\
\epsilon^{\Sigma}_{X} & \stackrel{\text{def}}{=} & \epsilon \\
(\hat{\sigma} \cdot \Sigma_1)^{\Sigma}_{X} & \stackrel{\text{def}}{=} & \hat{\sigma}^{\Sigma}_{X} \cdot \Sigma_1{}^{\Sigma}_{X} \\
(L \cdot \Sigma_1)^{\Sigma}_{X} & \stackrel{\text{def}}{=} & L \cdot \Sigma_1{}^{\Sigma}_{X} \\
X^{\Sigma}_{X} & \stackrel{\text{def}}{=} & \Sigma \\
Y^{\Sigma}_{X} & \stackrel{\text{def}}{=} & Y, \quad \text{si} \quad X \neq Y
\end{array}
$$

- Asignación de tipo

$$
\begin{array}{rcl}
\cdot^{\Sigma}_{X} & \stackrel{\text{def}}{=} & \cdot \\
(Y, \Theta)^{\Sigma}_{X} & \stackrel{\text{def}}{=} & Y^{\Sigma}_{X}, \Theta^{\Sigma}_{X}
\end{array}
$$

- Bloques de código tipado

$$
\begin{array}{rcl}
(\texttt{CODE}\langle \forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^{l}.B)^{\Sigma}_{X} & & \\
& \stackrel{\text{def}}{=} & \texttt{CODE}\langle \forall[\Theta']\Gamma\{\gamma\}^{\Sigma}_{X} \mid \mathbf{pc}\rangle^{l}.B\{\gamma\}^{\Sigma}_{X} \\
& & \text{con} \quad \gamma \text{ renombre } \Theta \text{ hacia } \Theta' \\
& & \text{y} \quad \Theta' \quad \text{fresh} \quad (\Theta' \not\subseteq FV(\Sigma) \cup \{X\})
\end{array}
$$

- Bloques de código

$$
\begin{aligned}
\texttt{halt}^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{halt} \\
(\texttt{jmp}\ v)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{jmp}\ v^{\Sigma}_{X} \\
(\texttt{jmpJP}\ L[\Sigma_1])^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{jmpJP}\ L[\Sigma_1{}^{\Sigma}_{X}] \\
(\texttt{aop}\ r_d, r_s, v)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{aop}\ r_d, r_s, v^{\Sigma}_{X} \\
(\texttt{bnz}\ r, v)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{bnz}\ r, v^{\Sigma}_{X} \\
(\texttt{mov}\ r, v)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{mov}\ r, v^{\Sigma}_{X} \\
(\texttt{ld}\ r_d, r_s[i])^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{ld}\ r_d, r_s[i] \\
(\texttt{st}\ r_d[i], r_s)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{st}\ r_d[i], r_s \\
(\texttt{push}\ L)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{push}\ L \\
(\texttt{salloc}\ i)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{salloc}\ i \\
(\texttt{sfree}\ i)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{sfree}\ i \\
(\texttt{sld}\ r_d, \texttt{sp}[i])^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{sld}\ r_d, \texttt{sp}[i] \\
(\texttt{sst}\ \texttt{sp}[i], r_s)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{sst}\ \texttt{sp}[i], r_s \\
(\texttt{sstNsl}\ \mathit{level}, \texttt{sp}[i], r_s)^{\Sigma}_{X} &\stackrel{\text{def}}{=} \texttt{sstNsl}\ \mathit{level}, \texttt{sp}[i], r_s
\end{aligned}
$$

2.3. Semántica Operacional

La semántica operacional de SecTAL se define en la figura 2.7. Es igual a aquella definida para SIFTAL, con el agregado de la instrucción `sstNsl`. Note que el efecto que produce dicha instrucción en ejecución es igual al efecto que produce la instrucción `sst`. Esto se debe, como explicamos anteriormente, a que `sstNsl` es una directiva de tipado y será reemplazada por `sst` antes de comenzar la ejecución. De forma similar, se puede observar que la directiva de tipado `jmpJP` L tiene el mismo efecto en ejecución que `jmp` v cuando $v = L$. Particularmente, la directiva de tipado `pushJP` no tiene ningún efecto en ejecución y puede ser eliminada una vez finalizado el chequeo de tipos.

Utilizamos semántica operacional del tipo estructural o small-step [PL81]. Decimos que una *configuración de máquina* Π evalúa en Π', si $\Pi \rightarrow \Pi'$. Usamos $\twoheadrightarrow$ para la clausura transitiva y reflexiva de $\rightarrow$.

Para ejecutar la instrucción `jmp` v, primero se obtiene la etiqueta del bloque de código de destino de v y luego el bloque de código desde el heap. Definimos la función $\hat{R}$ para obtener el valor de un operando v.

$$
\hat{R}(v) = \begin{cases} R(r), & \text{si } v = r \\ w, & \text{si } v = w \\ \hat{R}(v_1)[\Sigma], & \text{si } v = v_1[\Sigma] \end{cases}
$$

$$\boxed{(H, R, B) \longrightarrow \Pi}$$

donde si $B =$	entonces $\Pi =$	
`jmp` v	$(H, R, B'^{\overline{\Sigma}}_{\Theta})$ donde $\hat{R}(v) = L[\Sigma]$ y $H(L) = \texttt{CODE}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}'\rangle^l.B'$	SO_Jmp
`jmpJP` $L[\Sigma]$	(H, R, B'^{Σ}_{X}) donde $H(L) = \texttt{CODE}\langle\forall[X]\Gamma \mid \mathbf{pc}'\rangle^l.B'$	SO_JmpJP
`aop` $r_d, r_s, v; B'$	$(H, R[r_d := n], B')$ donde $n = \hat{R}(v) \oplus R(r_s)$	SO_Arith
`bnz` $r, v; B'$	(H, R, B') donde $R(r) = 0$	SO_Bnz1
`bnz` $r, v; B'$	$(H, R, B''^{\overline{\Sigma}}_{\Theta})$ donde $R(r) \neq 0, \hat{R}(v) = L[\overline{\Sigma}]$ y $H(L) = \texttt{CODE}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}''\rangle^l.B''$	SO_Bnz2
`mov` $r, v; B'$	$(H, R[r := \hat{R}(v)], B')$	SO_Mov
`ld` $r_d, r_s[i]; B'$	$(H, R[r_d := w_i], B')$ donde $R(r_s) = p$ y $H(p) = \langle w_0, \ldots, w_i, \ldots, w_n\rangle$	SO_Load
`st` $r_d[i], r_s; B'$	$(H[p := \langle w_0, \ldots, R(r_s), \ldots, w_n\rangle], R, B')$ donde $R(r_d) = p$ y $H(p) = \langle w_0, \ldots, w_i, \ldots, w_n\rangle$	SO_Store
`pushJP` $L; B'$	(H, R, B')	SO_Push
`salloc` $i; B'$	$(H, R[\texttt{sp} := \underbrace{ns \cdot \ldots \cdot ns}_{i} \cdot S], B')$ donde $R(\texttt{sp}) = S$	SO_Salloc
`sfree` $i; B'$	$(H, R[\texttt{sp} := S], B')$ donde $R(\texttt{sp}) = \hat{w}_1 \cdot \ldots \cdot \hat{w}_i \cdot S$	SO_Sfree
`sld` $r_d, \texttt{sp}[i]; B'$	$(H, R[r_d := w_i], B')$ donde $R(\texttt{sp}) = \hat{w}_0 \cdot \ldots \cdot \hat{w}_i \cdot S$	SO_Sld
`sst` $\texttt{sp}[i], r_s; B'$	$(H, R[\texttt{sp} := \hat{w}_0 \cdot \ldots \cdot \hat{w}_{i-1} \cdot R(r_s) \cdot S], B')$ donde $R(\texttt{sp}) = \hat{w}_0 \cdot \ldots \cdot \hat{w}_i \cdot S$	SO_Sst
`sstNsl` $level, \texttt{sp}[i], r_s; B'$	$(H, R[\texttt{sp} := \hat{w}_0 \cdot \ldots \cdot \hat{w}_{i-1} \cdot R(r_s) \cdot S], B')$ donde $R(\texttt{sp}) = \hat{w}_0 \cdot \ldots \cdot \hat{w}_i \cdot S$	SO_SstNsl

Figura 2.7: Semántica Operacional

2.4. Sistema de Tipos

En esta sección presentamos las reglas de tipado para SecTAL. Como se mencionó anteriormente, SecTAL es una variante de SIFTAL, con lo cual no es la intención de esta sección la de dar una definición completa, la misma puede encontrarse en [BCM04, MED06]. Aquí presentamos lo necesario para comprender el siguiente capítulo.

Para comenzar definiremos la siguiente notación. Si $\Gamma = \{r_1 : \sigma_1, \ldots, r_n : \sigma_n, \mathtt{sp} : \Sigma\}$, entonces $Dom(\Gamma)$ es el conjunto $\{r_1, \ldots, r_n, \mathtt{sp}\}$. Utilizaremos $\Gamma[r := \sigma]$ para representar el contexto de tipado de registros luego de una modificación a Γ con la asociación $r : \sigma$. Dado que $\sigma ::= \tau^l$, utilizaremos la notación $\sigma^{\sqcup l_1}$ que significa $\tau^{l \sqcup l_1}$. Definimos $\mathtt{label}(\tau^l) = l$.

Las reglas de tipado que se presentan a continuación, están organizadas dentro de los siguientes juicios de tipado:

$\Theta \rhd \tau$ ok	Tipo bien formado
$\Theta \rhd \sigma$ ok	Tipo de seguridad bien formado
$\Theta \rhd \Sigma$ ok	Tipo de pila bien formado
$\rhd \Psi$ ok	Tipo de heap bien formado
$\Theta \rhd \Gamma$ ok	Tipo de banco de registros bien formado
$\Theta \rhd \tau \leq \tau'$	Subtipado de tipos básicos
$\Theta \rhd \sigma \leq \sigma'$	Subtipado de tipos de seguridad
$\Theta \rhd \Gamma \leq \Gamma'$	Subtipado de bancos de registros
$\Theta \rhd_\Psi w : \sigma$ wval	Valores word bien tipados
$\Theta \mid \Gamma \rhd_\Psi v : \sigma$ opnd	Operando bien tipado
$\rhd_\Psi h : \sigma$ hval	Valor de heap bien tipado
$\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi B$ blk	Bloque de código bien tipado
$\rhd_\Psi S : \Sigma$ cstack	Pila de control bien tipada
$\rhd H : \Psi$ heap	Heap bien tipado
$\rhd_\Psi R : \Gamma$ regBank	Banco de registros bien tipado
$\rhd (H, R, B) : [\Psi, \Gamma, \mathbf{pc}]$ machConfig	Máquina bien tipada

Las reglas de tipado que determinan que un bloque de código B es bien tipado, bajo Θ, Γ, Ψ y el program-counter **pc**, se presentan en las figuras 2.8 y 2.9. Observando la regla [T-Halt] podemos ver que la instrucción `halt` es tratada como si fuera una instrucción `jmp` que salta a un junction point que finaliza el programa. Debido a esto, se espera que en el tope de la pila se encuentre la etiqueta *Halt*. Luego, el juicio $\Theta \rhd \Gamma$ ok verifica que Γ este bien formado con respecto a Θ. Las reglas para los tipos bien formados se muestran en la figura 2.12. Para que la instrucción `jmp` v se pueda tipar, el contexto de tipado de registros Γ debe ser compatible con el esperado por el

bloque de destino v. Esto se logra utilizando subtipado cuyas reglas se muestran en la figura 2.10. Además, un bloque de código solo puede saltar a otro bloque de código cuyo nivel de seguridad es igual o mayor. Un tratamiento similar tiene la instrucción `jmpJP` $L[\Sigma]$. Primero que nada, para poder saltar al bloque de código cuya etiqueta es L, L debe estar en el tope de la pila. Segundo, el contexto de tipado de registros Γ debe ser compatible con el esperado por el bloque de destino L. Esto incluye el pasaje de los junction point aun pendientes que se encuentran en $\Gamma(\texttt{sp})$. Los tipos variables de pila X, que aparecen en $\Psi(L)$ se instancian con Σ. Asumimos que los bloques de código que representan junction points tienen sólo un tipo variable de pila y aparece al final. De allí la condición $\Gamma'(\texttt{sp}) = \alpha_1 \cdot \ldots \cdot \alpha_n \cdot X$ en la hipótesis de la regla en cuestión. Finalmente, dado que el junction point baja el nivel del **pc**, el nivel de seguridad del bloque de código L es irrelevante. No existen restricciones sobre las operaciones aritméticas como se puede ver en la regla [T-Arith]. El nivel de seguridad del registro r_d, se obtiene de aplicar la *junta* entre el **pc**, y los niveles de seguridad de r_s y v. La directiva de tipado `pushJP` L agrega L al tope de la pila y luego se utiliza esta nueva pila para tipar el resto del bloque de código. La condición $\mathbf{pc} \sqsubseteq \mathbf{pc}'$ asegura que cuando el junction point L sea llamado, el nivel de seguridad de $\mathbf{pc}'$ en L no sea mas bajo que el nivel de seguridad actual. Con la instrucción `salloc` i simplemente se agregan i tipos nonsense a la pila, y se utiliza la nueva pila para tipar el resto del bloque de código. De forma similar, la instrucción `sfree` i remueve los primeros i componentes de la pila. Estos componentes no pueden ser etiquetas, ya que interferiría con la estructura de junction points (solo las directivas `jmpJP` y `pushJP` pueden manipular los junction points). El resto de las reglas de tipado siguen la misma forma. Las reglas [T-SstNs] y [T-SstNsl], merecen un comentario aparte ya que no están presentes en SIFTAL. La regla [T-SstNs] se utiliza para tipar la instrucción `sst` $\texttt{sp}[i], r_s$ cuando la posición i de la pila es un tipo nonsense. La regla [T-SstNsl] se utiliza para tipar la directiva de tipado `sstNsl` *level* $\texttt{sp}[i], r_s$, la cual guarda en la pila el registro r_2 con tipo *level* independientemente del tipo actual de r_2.

Las reglas de tipado para los valores, los operandos y los valores del heap se muestran en la figura 2.13. Como podemos ver las constantes reciben el tipo $int^{\perp}$. El tipo de los bloques de código se obtiene de Ψ. También contamos con una regla de subtipado que permite utilizar σ' donde σ es esperado siempre que $\sigma' < \sigma$. Luego, la regla [T-WvalInst] se utiliza para tipar valores con la forma $w[\Sigma]$. Lo mismo aplica para las reglas de los operandos. Finalmente, la regla [T-CBlk] se utiliza para tipar bloques de código con nivel de seguridad.

Las reglas de tipado de la pila de control, el heap, los registros y la configuración de máquina se presentan en la figura 2.11. Note que las reglas de la pila de control no requieren comentarios adicionales con la excepción de [T-ConSLbl], que indica que para tipar la pila las etiquetas de los junction points deben ser ignoradas. La regla [T-Heap] dice que para que el heap tipe cada etiqueta que pertenezca a su dominio, debe estar

asociada a un valor del heap tipado. De forma similar, la regla [T-RegBank] dice que para que el conjunto de registros tipen, cada uno de los registros, con excepción de `sp`, tiene que estar asociado a un valor tipado. Por último, como se puede ver en la regla [T-MachConfig], para tipar una configuración de máquina (H, R, B) se requiere que el heap, los registros, el bloque de código actual y la pila de control tipen respecto de Ψ, Γ y el nivel de seguridad del **pc**.

El siguiente teorema nos dice que para una *configuración de máquina* tipable siempre habrá una regla semántica que nos permita avanzar un paso en la ejecución o evaluación, o bien terminamos la ejecución.

Teorema 1 (progreso). Si $\triangleright\Pi : [\Psi, \Gamma, \mathbf{pc}]$ machConfig entonces existe Π' tal que $\Pi \longrightarrow \Pi'$, o Π tiene la forma de $(H, R, \texttt{halt})$ con $\Gamma(\texttt{sp}) = \mathit{Halt} \cdot \Sigma$.

Esquema de prueba. La demostración del **Teorema 1 (progreso)** es por análisis de casos sobre la derivación de tipos de $\triangleright\Pi : [\Psi, \Gamma, \mathbf{pc}]$ machConfig. □

Por otro lado, el siguiente teorema nos dice que luego de avanzar un paso en la ejecución, la *configuración de máquina* obtenida será tipable.

Teorema 2 (preservación). Si $\triangleright\Pi : \Omega$ machConfig y $\Pi \longrightarrow \Pi'$ entonces existe Ω' tal que $\triangleright\Pi' : \Omega'$ machConfig.

Esquema de prueba. La demostración del **Teorema 2 (preservación)** es por análisis de casos sobre la derivación de $\Pi \longrightarrow \Pi'$. □

2.5. Ejemplo

La figura 2.14 presenta un programa escrito en SecTAL para el cálculo del factorial. Consiste en cuatro bloques de código, a saber *fact*, *nonzero*, *cont* y *tp*. El cálculo del factorial comienza en el bloque de código $fact$ y se calcula sobre un valor positivo y privado almacenado en el registro r_2. Cuando la ejecución llega al bloque *nonzero*, las instrucciones `sst sp`[0], r_2 y `sst sp`[1], r_a preparan la pila para la llamada recursiva. Luego, cuando la instrucción `jmp` $fact[int^{\top} \cdot tp \cdot X]$ se ejecuta, se instancia el tipo variable de pila X para que la pila del bloque llamador coincida con la pila del bloque llamado. Finalmente, cuando la ejecución llega al bloque de código *cont* y los componentes de la pila son accedidos mediante la instrucción `sld`, los registros r_2 y r_a poseen los mismos tipos que cuando fueron almacenados en la pila anteriormente.

$$\boxed{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}$$

$$\frac{\Theta \rhd \Gamma \ \mathsf{ok} \quad \Gamma(\mathtt{sp}) = \mathit{Halt} \cdot \Sigma}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{halt} \ \mathsf{blk}} \quad [\mathsf{T-Halt}]$$

$$\frac{\begin{array}{c}\Theta \mid \Gamma \rhd_\Psi v : \mathtt{CODE}\langle\forall[\cdot]\Gamma' \mid \mathbf{pc}'\rangle^{l} \ \mathsf{opnd} \\ \Theta \rhd \mathtt{CODE}\langle\forall[\cdot]\Gamma' \mid \mathbf{pc}'\rangle \leq \mathtt{CODE}\langle\forall[\cdot]\Gamma \mid \mathbf{pc} \sqcup l\rangle\end{array}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{jmp} \ v \ \mathsf{blk}} \quad [\mathsf{T-Jmp}]$$

$$\frac{\begin{array}{c}\Theta \rhd \mathtt{CODE}\langle\forall[\cdot]{\Gamma'}^{\Sigma}_{X} \mid \mathbf{pc}'\rangle \leq \mathtt{CODE}\langle\forall[\cdot]\Gamma[\mathtt{sp} := \Sigma'] \mid l\rangle \\ \Gamma'(\mathtt{sp}) = \alpha_1 \cdot \ldots \alpha_n \cdot X \quad \Theta \rhd \Sigma \ \mathsf{ok} \quad \Gamma(\mathtt{sp}) = L \cdot \Sigma' \quad \Psi(L) = \mathtt{CODE}\langle\forall[X]\Gamma' \mid \mathbf{pc}'\rangle^{l}\end{array}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{jmpJP} \ L[\Sigma] \ \mathsf{blk}} \quad [\mathsf{T-JmpJP}]$$

$$\frac{\begin{array}{c}\Theta \mid \Gamma \rhd_\Psi r_s : \mathit{int}^{l_1} \ \mathsf{opnd} \qquad \Theta \mid \Gamma \rhd_\Psi v : \mathit{int}^{l_2} \ \mathsf{opnd} \\ \Theta \mid \Gamma[r_d := \mathit{int}^{\mathbf{pc} \sqcup l_1 \sqcup l_2}] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}\end{array}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{aop} \ r_d, r_s, v; B \ \mathsf{blk}} \quad [\mathsf{T-Arith}]$$

$$\frac{\begin{array}{c}\Theta \rhd \mathtt{CODE}\langle\forall[\cdot]\Gamma' \mid \mathbf{pc}'\rangle \leq \mathtt{CODE}\langle\forall[\cdot]\Gamma \mid \mathbf{pc} \sqcup l_1 \sqcup l_2\rangle \\ \Theta \mid \Gamma \rhd_\Psi r : \mathit{int}^{l_1} \ \mathsf{opnd} \quad \Theta \mid \Gamma \rhd_\Psi v : \mathtt{CODE}\langle\forall[\cdot]\Gamma' \mid \mathbf{pc}'\rangle^{l_2} \ \mathsf{opnd} \\ \Theta \mid \Gamma \mid \mathbf{pc} \sqcup l_1 \rhd_\Psi B \ \mathsf{blk}\end{array}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{bnz} \ r, v; B \ \mathsf{blk}}\text{,} \quad [\mathsf{T-CondBranch}]$$

$$\frac{\Theta \mid \Gamma \rhd_\Psi v : {\tau_1}^{l_1} \ \mathsf{opnd} \qquad \Theta \mid \Gamma[r := {\tau_1}^{\mathbf{pc} \sqcup l_1}] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{mov} \ r, v; B \ \mathsf{blk}} \quad [\mathsf{T-Mov}]$$

$$\frac{\begin{array}{c}\Theta \mid \Gamma \rhd_\Psi r_s : \langle\sigma_0, \ldots, \sigma_i, \ldots, \sigma_n\rangle^{l_1} \ \mathsf{opnd} \\ \Theta \mid \Gamma[r_d := \sigma_i^{\sqcup \mathbf{pc} \sqcup l_1}] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}\end{array}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{ld} \ r_d, r_s[i]; B \ \mathsf{blk}} \quad [\mathsf{T-Ld}]$$

Figura 2.8: Reglas de tipado para bloques de código (Parte 1)

$$\frac{\Theta \mid \Gamma \rhd_\Psi r_d : \langle \sigma_0, \ldots, \sigma_i, \ldots, \sigma_n \rangle^l \ \mathsf{opnd} \qquad \Theta \mid \Gamma \rhd_\Psi r_s : \sigma_i \ \mathsf{opnd} \qquad \mathbf{pc} \sqcup l \sqsubseteq \mathtt{label}(\sigma_i) \qquad \Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{st}\ r_d[i], r_s; B \ \mathsf{blk}} \quad [\mathsf{T-St}]$$

$$\frac{\Psi(L) = \mathtt{CODE}\langle \forall [\Theta']\Gamma' \mid \mathbf{pc}' \rangle^{l'} \qquad \mathbf{pc} \sqsubseteq \mathbf{pc}' \qquad \Gamma(\mathtt{sp}) = \Sigma \qquad \Theta \mid \Gamma[\mathtt{sp} := L \cdot \Sigma] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{pushJP}\ L; B \ \mathsf{blk}} \quad [\mathsf{T-Push}]$$

$$\frac{\Gamma(\mathtt{sp}) = \Sigma \qquad \Theta \mid \Gamma[\mathtt{sp} := \overbrace{ns \cdot \ldots \cdot ns}^{i} \cdot \Sigma] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{salloc}\ i; B \ \mathsf{blk}} \quad [\mathsf{T-Salloc}]$$

$$\frac{\Gamma(\mathtt{sp}) = \hat{\sigma}_1 \cdot \ldots \hat{\sigma}_i \cdot \Sigma \qquad \Theta \mid \Gamma[\mathtt{sp} := \Sigma] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{sfree}\ i; B \ \mathsf{blk}} \quad [\mathsf{T-Free}]$$

$$\frac{\Gamma(\mathtt{sp}) = \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_{i-1} \cdot \tau^l \cdot \Sigma \qquad \Theta \mid \Gamma[r_d := \tau^{\mathbf{pc} \sqcup l}] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{sld}\ r_d, \mathtt{sp}[i]; B \ \mathsf{blk}} \quad [\mathsf{T-Sld}]$$

$$\frac{\Gamma(\mathtt{sp}) = \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_{i-1} \cdot ns \cdot \Sigma \qquad \Theta \mid \Gamma \rhd_\Psi r_s : \tau^l \ \mathsf{opnd} \qquad \Theta \mid \Gamma[\mathtt{sp} := \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_{i-1} \cdot \tau^{\mathbf{pc} \sqcup l} \cdot \Sigma] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{sst}\ \mathtt{sp}[i], r_s; B \ \mathsf{blk}} \quad [\mathsf{T-SstNs}]$$

$$\frac{\Gamma(\mathtt{sp}) = \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_{i-1} \cdot {\tau_1}^{l_1} \cdot \Sigma \qquad \mathbf{pc} \sqcup l_2 \sqsubseteq l_1 \qquad \Theta \mid \Gamma \rhd_\Psi r_s : {\tau_2}^{l_2} \ \mathsf{opnd} \qquad \Theta \mid \Gamma[\mathtt{sp} := \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_{i-1} \cdot {\tau_2}^{l_1} \cdot \Sigma] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{sst}\ \mathtt{sp}[i], r_s; B \ \mathsf{blk}} \quad [\mathsf{T-Sst}]$$

$$\frac{\Gamma(\mathtt{sp}) = \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_{i-1} \cdot ns \cdot \Sigma \qquad \Theta \mid \Gamma \rhd_\Psi r_s : \tau^l \ \mathsf{opnd} \qquad \Theta \mid \Gamma[\mathtt{sp} := \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_{i-1} \cdot \tau^{level} \cdot \Sigma] \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{sstNsl}\ level, \mathtt{sp}[i], r_s; B \ \mathsf{blk}} \quad [\mathsf{T-SstNsl}]$$

Figura 2.9: Reglas de tipado para bloques de código (Parte 2)

$\boxed{\Theta \rhd \tau \leq \tau'}$ $\boxed{\Theta \rhd \sigma \leq \sigma'}$ $\boxed{\Theta \rhd \Gamma \leq \Gamma'}$

$$\Theta \rhd int \leq int \qquad [\mathsf{ST}-\mathsf{Int}]$$

$$\frac{\Theta \rhd \tau_1 \leq \tau_2 \quad l_1 \sqsubseteq l_2}{\Theta \rhd {\tau_1}^{l_1} \leq {\tau_2}^{l_2}} \qquad [\mathsf{ST}-\mathsf{Latt}]$$

$$\frac{\Theta, \Theta'' \rhd \Gamma'\{\gamma\} \leq \Gamma\{\gamma\} \quad \gamma \text{ renombre } \Theta' \text{ to } \Theta'' \ \mathbf{pc}' \sqsubseteq \mathbf{pc}}{\Theta \rhd \mathtt{CODE}\langle\forall[\Theta']\Gamma \mid \mathbf{pc}\rangle \leq \mathtt{CODE}\langle\forall[\Theta']\Gamma' \mid \mathbf{pc}'\rangle} \qquad [\mathsf{ST}-\mathsf{Code}]$$

$$\frac{m \geq n \qquad l_i \sqsubseteq l'_i,\ i \in 1..n \qquad \Theta \rhd \tau_j \text{ ok},\ j \in 1..m \qquad \Theta \rhd \Sigma \text{ ok}}{\Theta \rhd \{r_1 : \tau_1^{l_1}, \ldots, r_m : \tau_m^{l_m}, \mathsf{sp} : \Sigma\} \leq \{r_1 : \tau_1^{l'_1}, \ldots, r_n : \tau_n^{l'_n}, \mathsf{sp} : \Sigma\}} \qquad [\mathsf{ST}-\mathsf{RegBank}]$$

Figura 2.10: Reglas de subtipado

$\boxed{\rhd_\Psi S : \Sigma \text{ cstack}}$

$$\rhd_\Psi \epsilon : \epsilon \text{ cstack} \qquad [\mathsf{T}-\mathsf{ConSNil}] \qquad \frac{\rhd_\Psi S : \Sigma \text{ cstack} \quad \cdot \rhd_\Psi w : \sigma \text{ wval}}{\rhd_\Psi w \cdot S : \sigma \cdot \Sigma \text{ cstack}} \qquad [\mathsf{T}-\mathsf{ConSCons}]$$

$$\frac{\rhd_\Psi S : \Sigma \text{ cstack}}{\rhd_\Psi S : L \cdot \Sigma \text{ cstack}} \qquad [\mathsf{T}-\mathsf{ConSLbl}] \qquad \frac{\rhd_\Psi S : \Sigma \text{ cstack}}{\rhd_\Psi ns \cdot S : ns \cdot \Sigma \text{ cstack}} \qquad [\mathsf{T}-\mathsf{ConSConsNs}]$$

$\boxed{\rhd H : \Psi \text{ heap}}$ $\boxed{\rhd_\Psi R : \Gamma \text{ regBank}}$ $\boxed{\rhd (H, R, B) \text{ machConfig}}$

$$\frac{Dom(H) = Dom(\Psi) \quad (\forall \ell \in Dom(H)) \rhd_\Psi H(\ell) : \Psi(\ell) \text{ hval} \quad \rhd \Psi \text{ ok}}{\rhd H : \Psi \text{ heap}} \qquad [\mathsf{T}-\mathsf{Heap}]$$

$$\frac{(\forall r \in Dom(\Gamma) \setminus \{\mathsf{sp}\}) \cdot \rhd_\Psi R(r) : \Gamma(r) \text{ wval}}{\rhd_\Psi R : \Gamma \text{ regBank}} \qquad [\mathsf{T}-\mathsf{RegBank}]$$

$$\frac{\rhd H : \Psi \text{ heap} \quad \rhd_\Psi R : \Gamma \text{ regBank} \quad \cdot \mid \Gamma \mid \mathbf{pc} \rhd_\Psi B \text{ blk} \quad \rhd_\Psi R(\mathsf{sp}) : \Gamma(\mathsf{sp}) \text{ cstack}}{\rhd (H, R, B) : [\Psi, \Gamma, \mathbf{pc}] \text{ machConfig}} \qquad [\mathsf{T}-\mathsf{MachConfig}]$$

Figura 2.11: Reglas de tipado de la pila de control, heaps, registros y configuración de máquina

$$\boxed{\Theta \rhd \tau \ \mathsf{ok}}$$

$$\Theta \rhd \mathit{int} \ \mathsf{ok} \qquad \frac{\Theta \rhd \sigma_i \ \mathsf{ok} \quad i \in \{1, \ldots, n\}}{\Theta \rhd \langle \sigma_1, \ldots, \sigma_n \rangle \ \mathsf{ok}} \qquad \frac{\Theta \cup \Theta' \rhd \Gamma \ \mathsf{ok}}{\Theta \rhd \mathtt{CODE}\langle \forall [\Theta']\Gamma \mid \mathbf{pc} \rangle \ \mathsf{ok}}$$

$$\boxed{\Theta \rhd \sigma \ \mathsf{ok}} \qquad \boxed{\Theta \rhd \Sigma \ \mathsf{ok}}$$

$$\frac{\Theta \rhd \tau \ \mathsf{ok} \quad l \in \mathfrak{L}_{\mathbf{sec}}}{\Theta \rhd \tau^l \ \mathsf{ok}} \qquad \frac{FV(\Sigma) \subseteq \Theta}{\Theta \rhd \Sigma \ \mathsf{ok}}$$

$$\boxed{\Theta \rhd \Psi \ \mathsf{ok}} \qquad \boxed{\Theta \rhd \Gamma \ \mathsf{ok}}$$

$$\frac{(\forall \ell \in \mathit{Dom}(\Psi)) \cdot \rhd \Psi(\ell) \ \mathsf{ok}}{\rhd \Psi \ \mathsf{ok}} \qquad \frac{\Theta \rhd \sigma_1 \ \mathsf{ok} \quad \ldots \Theta \rhd \sigma_n \ \mathsf{ok} \quad \Theta \rhd \Sigma \ \mathsf{ok}}{\Theta \rhd \{r_1 : \sigma_1, \ldots, r_n : \sigma_n, \mathsf{sp} : \Sigma\} \ \mathsf{ok}}$$

Figura 2.12: Tipos bien formados

2.6. No-Interferencia

Como se explicó en el capítulo anterior, tenemos que asegurar que un observador con acceso a cierto nivel de seguridad, llamemos a este nivel ζ, no pueda obtener ni inferir información de niveles de seguridad más altos. Por lo tanto, tenemos que demostrar que para cualquier ejecución de un programa SecTAL que pase el chequeo de tipos, los valores con nivel de seguridad mas altos que ζ, no interfieran sobre los valores con igual o menor nivel de seguridad que ζ. De esta forma decimos que un valor con nivel de seguridad *low* es aquel que es menor o igual al nivel de seguridad al que tiene acceso el observador, ζ. Y un valor con nivel de seguridad *high* es aquel que es mayor a ζ.

Dado que un observador sólo puede obtener información de valores *low*, dos *configuraciones de máquina* SecTAL se dicen *indistinguibles* si los valores *low* finales de ambas configuraciones son los mismos. Antes de formalizar la noción de no-interferencia en SecTAL, definimos *indistinguibilidad* de configuraciones de máquina con respecto al nivel de seguridad al que tiene acceso el observador, ζ. Llamaremos a esto *indistinguibilidad-ζ*.

2.6.1. Indistinguibilidad-ζ

Definir la noción de *indistinguibilidad-ζ* entre *configuraciones de máquina*, requiere primero que se defina para cada uno de los componentes de la máquina, a saber, el *heap*, los *registros* (incluidos la pila de control) y el bloque de código que se esta ejecutando. Comenzaremos con la pila de control.

Cuando tomamos dos ejecuciones de un mismo programa, mientras no haya bifurcaciones, ambas ejecuciones realizan exactamente los mismos pasos, las pilas tienen el

$$\boxed{\Theta \rhd_\Psi w : \sigma \text{ wval}}$$

$$\Theta \rhd_\Psi i : int^\perp \text{ wval} \qquad [\mathsf{T}-\mathsf{IntLit}]$$

$$\frac{\Theta \rhd \Psi(L) \text{ ok}}{\Theta \rhd_\Psi L : \Psi(L) \text{ wval}} \qquad [\mathsf{T}-\mathsf{CodeLbl}]$$

$$\frac{\Theta \rhd_\Psi w : \sigma \text{ wval} \qquad \Theta \rhd \sigma \leq \sigma'}{\Theta \rhd_\Psi w : \sigma' \text{ wval}} \qquad [\mathsf{T}-\mathsf{WvalSub}]$$

$$\frac{\Theta \rhd_\Psi w : \texttt{CODE}\langle\forall[X,\Theta']\Gamma \mid \mathbf{pc}\rangle^l \text{ wval} \qquad \Theta \rhd \Sigma \text{ ok} \qquad X,\Theta' \cap FV(\Sigma) = \emptyset}{\Theta \rhd_\Psi w[\Sigma] : \texttt{CODE}\langle\forall[\Theta']\Gamma^\Sigma_X \mid \mathbf{pc}\rangle^l \text{ wval}} \qquad [\mathsf{T}-\mathsf{WvalInst}]$$

$$\boxed{\Theta \mid \Gamma \rhd_\Psi v : \sigma \text{ opnd}}$$

$$\frac{\Theta \rhd \Gamma \text{ ok} \quad \Theta \rhd_\Psi w : \sigma \text{ wval}}{\Theta \mid \Gamma \rhd_\Psi w : \sigma \text{ opnd}} \qquad [\mathsf{T}-\mathsf{WordOp}]$$

$$\frac{\Theta \rhd \Gamma \text{ ok}}{\Theta \mid \Gamma \rhd_\Psi r : \Gamma(r) \text{ opnd}} \qquad [\mathsf{T}-\mathsf{RegOp}]$$

$$\frac{\Theta \mid \Gamma \rhd_\Psi v : \sigma \text{ opnd} \quad \Theta \rhd \sigma \leq \sigma'}{\Theta \mid \Gamma \rhd_\Psi v : \sigma' \text{ opnd}} \qquad [\mathsf{T}-\mathsf{OpndSub}]$$

$$\frac{\Theta \mid \Gamma \rhd_\Psi v : \texttt{CODE}\langle\forall[X,\Theta']\Gamma \mid \mathbf{pc}\rangle^l \text{ opnd} \quad \Theta \rhd \Sigma \text{ ok} \quad X,\Theta' \cap FV(\Sigma) = \emptyset}{\Theta \mid \Gamma \rhd_\Psi v[\Sigma] : \texttt{CODE}\langle\forall[\Theta']\Gamma^\Sigma_X \mid \mathbf{pc}\rangle^l \text{ opnd}} \qquad [\mathsf{T}-\mathsf{OpndInst}]$$

$$\boxed{\rhd_\Psi h : \sigma \text{ hval}}$$

$$\frac{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi B \text{ blk}}{\rhd_\Psi \texttt{CODE}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^l.B : \texttt{CODE}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^l \text{ hval}} \qquad [\mathsf{T}-\mathsf{CBlk}]$$

Figura 2.13: Reglas de tipado de los valores, los operandos y valores del heap

fact:

$\texttt{CODE}\langle\forall[X]\{r_1 : int^\top, r_2 : int^\top, r_a : tp, \texttt{sp} : X\}|^\top\rangle^\top$

$\texttt{bnz}\ r_2,\ nonzero[X]$

$\texttt{mov}\ r_1,\ 1$

$\texttt{jmp}\ r_a$

nonzero:

$\texttt{CODE}\langle\forall[X]\{r_1 : int^\top, r_2 : int^\top, r_3 : int^\top, r_a : tp, \texttt{sp} : X\}|^\top\rangle^\top$

$\texttt{sub}\ r_3,\ r_2,\ 1$

$\texttt{salloc}\ 2$

$\texttt{sst}\ \texttt{sp}[0],\ r_2$

$\texttt{sst}\ \texttt{sp}[1],\ r_a$

$\texttt{mov}\ r_2,\ r_3$

$\texttt{mov}\ r_a,\ cont[X]$

$\texttt{jmp}\ fact[int^\top.tp.X]$

cont:

$\texttt{CODE}\langle\forall[X]\{r_1 : int^\top, r_2 : int^\top, r_a : tp, \texttt{sp} : int^\top.tp.X\}|^\top\rangle^\top$

$\texttt{sld}\ r_2,\ \texttt{sp}[0]$

$\texttt{sld}\ r_a,\ \texttt{sp}[1]$

$\texttt{sfree}\ 2$

$\texttt{mul}\ r_1,\ r2,\ r1$

$\texttt{jmp}\ r_a$

tp: $\texttt{CODE}\langle\forall[]\{r1 : int^\top, \texttt{sp} : X\}|^\top\rangle^\top$

$\bot$: nivel de seguridad público $\top$: nivel de seguridad privado

Figura 2.14: Ejemplo de un programa en SecTAL

mismo tamaño y los mismos tipos (*high* o *low*) en las mismas posiciones. Si hay bifurcaciones que dependen de valores *low*, dado que consideramos que los valores de entrada *low* son los mismos para las dos ejecuciones, seguimos manteniendo pilas iguales en ambas ejecuciones. En este caso decimos que las pilas son *indistinguibles* en *nivel low*. Esta noción es formalizada en la figura 2.15.

$$\boxed{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackLow}}$$

$$\rhd_{\Psi_1,\Psi_2} \epsilon \approx_\zeta \epsilon : \epsilon \wedge \epsilon \ \mathsf{cstackLow} \qquad [\mathsf{EqESLAxiom}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackLow} \quad \rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \sigma_1 \wedge \sigma_2 \ \mathsf{wval}}{\rhd_{\Psi_1,\Psi_2} w_1 \cdot S_1 \approx_\zeta w_2 \cdot S_2 : \sigma_1 \cdot \Sigma_1 \wedge \sigma_2 \cdot \Sigma_2 \ \mathsf{cstackLow}} \qquad [\mathsf{EqESLLLHH}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackLow}}{\rhd_{\Psi_1,\Psi_2} ns \cdot S_1 \approx_\zeta ns \cdot S_2 : ns \cdot \Sigma_1 \wedge ns \cdot \Sigma_2 \ \mathsf{cstackLow}} \qquad [\mathsf{EqESLNonsense}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackLow} \quad \Psi_1(L) = \Psi_2(L) = \mathtt{CODE}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^l \quad \mathbf{pc} \sqcup l \sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge L \cdot \Sigma_2 \ \mathsf{cstackLow}} \qquad [\mathsf{EqESLSynch}]$$

$$\frac{\begin{array}{c}\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackLow} \qquad \Psi_1(L_1) = \mathtt{CODE}\langle\forall[\Theta_1]\Gamma_1 \mid \mathbf{pc}_1\rangle^{l_1} \quad \mathbf{pc}_1 \sqcup l_1 \not\sqsubseteq \zeta \\ \Psi_2(L_2) = \mathtt{CODE}\langle\forall[\Theta_2]\Gamma_2 \mid \mathbf{pc}_2\rangle^{l_2} \quad \mathbf{pc}_2 \sqcup l_2 \not\sqsubseteq \zeta\end{array}}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L_1 \cdot \Sigma_1 \wedge L_2 \cdot \Sigma_2 \ \mathsf{cstackLow}} \qquad [\mathsf{EqESLHighSynch}]$$

Figura 2.15: Pila de control indistinguibles en nivel low

La regla más simple, EqESLAxiom, nos dice que dos pilas son *indistinguibles* en *nivel low* si están vacías. La regla EqESLLLHH nos dice que dos pilas cuyos topes poseen *valores*, w_1 y w_2 sin *indistinguibles* si w_1 y w_2 son *indistinguibles* entre sí. Mas adelante en esta sección mostramos las condiciones para que dos *valores* sean *indistinguibles*. Las demás reglas siguen la misma forma, salvo EqESLHighSynch que aplica para cuando, estando con un **pc** *low*, la regla de tipado `pushJP` L, guarda en la pila un label L con tipo *high*.

Por otro lado, si encontramos una bifurcación que depende de un valor *high*, las ejecuciones pueden tomar caminos diferentes, o sea, ramas diferentes de la bifurcación. Esto, como mencionamos anteriormente, se debe a que los valores de entrada *high* pueden variar entre diferentes ejecuciones de un mismo programa. Como consecuencia de esto, las pilas de control pueden comenzar a variar dado que los programas ejecutan diferentes instrucciones. En este caso, decimos que las pilas son *indistinguibles* en *nivel high* y lo formalizamos en la figura 2.16. Note que la regla EqESHAxiom señala que dos

pilas *indistinguibles* en *nivel low*, también lo son en *nivel high*.

Tenemos que aclarar cuando dos *valores* son *indistinguibles*, formalizado como: $\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \sigma_1 \wedge \sigma_2$ wval y utilizado en la regla EqESLLLHH. En principio, damos la siguiente definición. Dos *valores* son *indistinguibles* en *nivel low* si σ_1 y σ_2 son *low* entonces $\sigma_1 = \sigma_2$ y $w_1 = w_2$. Y son *indistinguibles* en *nivel high* si σ_1 y σ_2 son *high* (note que esta definición es equivalente a la dada en el siguiente capítulo: Definición 1 (equivalencia de estados)). Analicemos el siguiente ejemplo donde B es el bloque de código inicial y $\sigma_X = \texttt{CODE}\langle\forall[\Theta]\{\texttt{sp} : X\} \mid \mathbf{pc}\rangle^\bot$.

$$
\begin{array}{lll}
 & & L1 \quad \texttt{CODE}\langle\forall[X]\{\texttt{r1} : \sigma_X, \texttt{sp} : X\} \mid \top\rangle^\top \\
B = & \texttt{push}\ L_{JP} & \quad\ \texttt{jmpJP}\ L_{JP}[\Sigma_2] \\
 & \texttt{bnz r}, L1 & \\
 & \texttt{jmpJP}\ L_{JP}[\Sigma_1] & L_{JP} \quad \texttt{CODE}\langle\forall[Y]\{\texttt{r1} : \sigma_Y, \texttt{sp} : Y\} \mid \bot\rangle^\bot \\
 & & \quad\ \ldots
\end{array}
$$

Tomemos dos ejecuciones de este programa donde las *configuraciones de máquina* iniciales cumplen las siguientes condiciones:

1. Ambas asignan un valor *low* al **pc**,
2. ambas asignan pilas *indistinguibles* en *nivel low* a `sp`,
3. el valor asignado al registro *high* `r` para una de las *configuraciones de máquina* es 0 y para la otra es 1.

Al encontrar la instrucción `bnz`, una de las ejecuciones saltará hacia *L1*, mientras que la otra continuará con la siguiente instrucción. Cuando ambas ejecuciones alcancen el *junction point* L_{JP}, se "sincronizarán" y volverán a estar en nivel *low*, dado que el *junction point* baja el **pc** a *low*. Note que en este punto de la ejecución la *configuración de máquina* de cada una de las ejecuciones se diferencia en el tipo del registro `r1`, dado que X ha sido instanciada con tipos diferentes (Σ_1 y Σ_2). El hecho de utilizar tipos polimórficos hace que la definición tentativa dada antes no sea suficiente, ya que como demuestra el ejemplo, los *valores* del registro `r1` no son *indistinguibles* en *nivel low*. Por lo tanto, la definición de indistinguibilidad entre *valores* debe considerar que sus tipos pueden diferir debido a que los tipos variables de sus pilas han sido instanciados. Además, los tipos de pila con los que se instancian los tipos variable de pila, deben ser *indistinguibles* en *nivel low*. Por esta razón, para poder finalmente formalizar la definición $\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \sigma_1 \wedge \sigma_2$ wval, primero tenemos que presentar la noción de *indistinguibilidad* en *nivel low* para los *tipos con nivel de seguridad* y los *tipo de la pila de control*. La figura 2.17 define dicha noción.

Informalmente, dos *tipos con nivel de seguridad* σ_1 y σ_2 son *indistinguibles* en *nivel low* si existe algún *tipo con nivel de seguridad* σ y las sustituciones γ_1 y γ_2 sobre los tipos variables de pila tal que $\sigma_1 = \sigma\{\gamma_1\}$, $\sigma_2 = \sigma\{\gamma_2\}$, y γ_1 y γ_2 asignan pilas

$$\boxed{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh}}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackLow}}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh}} \quad [\mathsf{EqESHAxiom}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh} \ \ l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} w \cdot S_1 \approx_\zeta S_2 : \tau^l \cdot \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh}} \quad [\mathsf{EqESHLeft}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh} \ \ l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta w \cdot S_2 : \Sigma_1 \wedge \tau^l \cdot \Sigma_2 \ \mathsf{cstackHigh}} \quad [\mathsf{EqESHRight}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh}}{\rhd_{\Psi_1,\Psi_2} ns \cdot S_1 \approx_\zeta S_2 : ns \cdot \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh}} \quad [\mathsf{EqESHLeftNs}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh}}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta ns \cdot S_2 : \Sigma_1 \wedge ns \cdot \Sigma_2 \ \mathsf{cstackHigh}} \quad [\mathsf{EqESHRightNs}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh} \ \ l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} w_1 \cdot S_1 \approx_\zeta ns \cdot S_2 : \tau^l \cdot \Sigma_1 \wedge ns \cdot \Sigma_2 \ \mathsf{cstackHigh}} \quad [\mathsf{EqESLHighNs}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh} \ \ l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} ns \cdot S_1 \approx_\zeta w_2 \cdot S_2 : ns \cdot \Sigma_1 \wedge \tau^l \cdot \Sigma_2 \ \mathsf{cstackHigh}} \quad [\mathsf{EqESLNsHigh}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh} \ \ \Psi_1(L) = \mathtt{CODE}\langle \forall[\Theta]\Gamma \mid \mathbf{pc} \rangle^l \ \ \mathbf{pc} \sqcup l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh}} \quad [\mathsf{EqESHLeftSynch}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{cstackHigh} \ \ \Psi_2(L) = \mathtt{CODE}\langle \forall[\Theta]\Gamma \mid \mathbf{pc} \rangle^l \ \ \mathbf{pc} \sqcup l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge L \cdot \Sigma_2 \ \mathsf{cstackHigh}} \quad [\mathsf{EqESHRightSynch}]$$

Figura 2.16: Pila de control indistinguibles en nivel high

indistinguibles en *nivel low* a las mismas variables. Esto mismo aplica para la noción de *indistinguibilidad* en *nivel low* para los tipos de pila. Tenga en cuenta que:

$\triangleright_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2$ cstackLow implica $\triangleright_{\Psi_1,\Psi_2} \Sigma_1 \approx_\zeta \Sigma_2$ stackTypeEq

Finalmente nos queda definir la noción de indistinguibilidad para los *valores*, *valores del heap*, *heap*, *registros* y *bloques de código*. Los mismos pueden verse en la figura 2.18 y 2.19. Note que en dichas figuras utilizaremos la siguiente notación: dada una secuencia de conjuntos $A, B, \ldots, Z$ y una operación $\oplus$ sobre los conjuntos, utilizaremos $Dom_\oplus(A, B, \ldots, Z)$ para decir $Dom(A) \oplus Dom(B) \oplus \ldots \oplus Dom(Z)$.

Por último, damos la definición de indistinguibilidad-ζ de *configuraciones de máquina*. Para que dos configuraciones de máquina sean indistinguibles, necesitamos que ambas sean tipables, y sus heaps y registros sean indistinguibles. Además, si el **pc** esta en nivel *low*, como mencionamos antes, ambos programas ejecutan las mismas instrucciones, con lo cual, sus bloques de código y sus pilas son *indistinguibles* en *nivel low*. Si el **pc** esta en nivel *high*, entonces sus pilas deben ser *indistinguibles* en *nivel high*.

Definición 1 (indistinguibilidad-ζ de configuraciones de máquina). Asumamos dos configuraciones de máquina $\Pi_i = (H_i, R_i, B_i)$ y sus tipos $\Omega_i = [\Psi_i, \Gamma_i, \mathbf{pc}_i]$ para $i \in \{1, 2\}$. El juicio $\triangleright \Pi_1 \approx_\zeta \Pi_2 : \Omega_1 \wedge \Omega_2$ machConfig se cumple si y solo si:

1. $\triangleright \Pi_1 : \Omega_1$ machConfig y $\triangleright \Pi_2 : \Omega_2$ machConfig
2. $\triangleright H_1 \approx_\zeta H_2 : \Psi_1 \wedge \Psi_2$ heap
3. $\triangleright_{\Psi_1,\Psi_2} R_1 \approx_\zeta R_2 : \Gamma_1 \wedge \Gamma_2$ regBank
4. either
   (a) $\mathbf{pc}_1 = \mathbf{pc}_2 \sqsubseteq \zeta$ y $\triangleright_{\Psi_1,\Psi_2} B_1 \approx_\zeta B_2$ code y $\triangleright_{\Psi_1,\Psi_2} R_1(\mathtt{sp}) \approx_\zeta R_2(\mathtt{sp}) : \Gamma_1(\mathtt{sp}) \wedge \Gamma_2(\mathtt{sp})$ cstackLow, o
   (b) $\mathbf{pc}_1 \not\sqsubseteq \zeta$ y $\mathbf{pc}_2 \not\sqsubseteq \zeta$ y $\triangleright_{\Psi_1,\Psi_2} R_1(\mathtt{sp}) \approx_\zeta R_2(\mathtt{sp}) : \Gamma_1(\mathtt{sp}) \wedge \Gamma_2(\mathtt{sp})$ cstackHigh.

2.6.2. Teorema de No-Interferencia

En esta sección presentamos los lemas necesarios para formular el teorema de no-interferencia para SecTAL. Como mencionamos anteriormente, tomamos dos ejecuciones de un mismo programa donde los valores de entrada *low* son los mismos y los valores de entrada *high* pueden ser diferentes. Ambas ejecuciones comienzan con configuraciones de máquina indistinguibles. Además, el **pc** inicial es *low* y la pila de control esta inicializada con la etiqueta *Halt* en el tope. Necesitamos dicha etiqueta para determinar cuando un programa finaliza. Como se puede ver en la regla de tipado T-Halt, la etiqueta *Halt* solo puede ser consumida por la instrucción `halt`.

$$\boxed{\rhd_{\Psi_1,\Psi_2}\sigma_1 \approx_\zeta \sigma_2 \ \mathsf{secTypeEq}}$$

El juicio anterior se cumple si existen las sustituciones γ_1 y γ_2, y el tipo σ tal que

1. $Dom(\gamma_1) = Dom(\gamma_2) = FV(\sigma)$
2. $\sigma_1 = \sigma\{\gamma_1\}$ y $\sigma_2 = \sigma\{\gamma_2\}$, y
3. por cada $X \in Dom(\gamma_1)$, $\rhd_{\Psi_1,\Psi_2} X\{\gamma_1\} \approx_\zeta X\{\gamma_2\}$ $\mathsf{stackTypeEq}$.

$$\boxed{\rhd_{\Psi_1,\Psi_2}\Sigma_1 \approx_\zeta \Sigma_2 \ \mathsf{stackTypeEq}}$$

$\rhd_{\Psi_1,\Psi_2}\epsilon \approx_\zeta \epsilon$ $\mathsf{stackTypeEq}$

$\rhd_{\Psi_1,\Psi_2} X \approx_\zeta X$ $\mathsf{stackTypeEq}$

$$\frac{\rhd_{\Psi_1,\Psi_2}\Sigma_1 \approx_\zeta \Sigma_2 \ \mathsf{stackTypeEq} \quad l \sqsubseteq \zeta \quad \rhd_{\Psi_1,\Psi_2} {\tau_1}^l \approx_\zeta {\tau_2}^l \ \mathsf{secTypeEq}}{\rhd_{\Psi_1,\Psi_2} {\tau_1}^l \cdot \Sigma_1 \approx_\zeta {\tau_2}^l \cdot \Sigma_2 \ \mathsf{stackTypeEq}} \quad [\mathsf{EqESLLLHH}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2}\Sigma_1 \approx_\zeta \Sigma_2 \ \mathsf{stackTypeEq} \quad l_1 \not\sqsubseteq \zeta \quad l_1 \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} {\tau_1}^{l_1} \cdot \Sigma_1 \approx_\zeta {\tau_2}^{l_2} \cdot \Sigma_2 \ \mathsf{stackTypeEq}} \quad [\mathsf{EqESLHigh}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2}\Sigma_1 \approx_\zeta \Sigma_2 \ \mathsf{stackTypeEq}}{\rhd_{\Psi_1,\Psi_2} ns \cdot \Sigma_1 \approx_\zeta ns \cdot \Sigma_2 \ \mathsf{stackTypeEq}} \quad [\mathsf{EqESLNonsense}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2}\Sigma_1 \approx_\zeta \Sigma_2 \ \mathsf{stackTypeEq} \quad l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} \tau^l \cdot \Sigma_1 \approx_\zeta ns \cdot \Sigma_2 \ \mathsf{stackTypeEq}} \quad [\mathsf{EqESLHighNs}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2}\Sigma_1 \approx_\zeta \Sigma_2 \ \mathsf{stackTypeEq} \quad l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} ns \cdot \Sigma_1 \approx_\zeta \tau^l \cdot \Sigma_2 \ \mathsf{stackTypeEq}} \quad [\mathsf{EqESLNsHigh}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2}\Sigma_1 \wedge \Sigma_2 \ \mathsf{stackTypeEq} \quad \Psi_1(L) = \Psi_2(L) = \mathtt{CODE}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^l \quad \mathbf{pc} \sqcup l \sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} L \cdot \Sigma_1 \approx_\zeta L \cdot \Sigma_2 \ \mathsf{stackTypeEq}} \quad [\mathsf{EqESLSynch}]$$

$$\frac{\rhd_{\Psi_1,\Psi_2}\Sigma_1 \approx_\zeta \Sigma_2 \ \mathsf{stackTypeEq} \qquad \Psi_1(L_1) = \mathtt{CODE}\langle\forall[\Theta_1]\Gamma_1 \mid \mathbf{pc}_1\rangle^{l_1} \quad \mathbf{pc}_1 \sqcup l_1 \not\sqsubseteq \zeta \quad \Psi_2(L_2) = \mathtt{CODE}\langle\forall[\Theta_2]\Gamma_2 \mid \mathbf{pc}_2\rangle^{l_2} \quad \mathbf{pc}_2 \sqcup l_2 \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} L_1 \cdot \Sigma_1 \approx_\zeta L_2 \cdot \Sigma_2 \ \mathsf{stackTypeEq}} \quad [\mathsf{EqESLHighSynch}]$$

Figura 2.17: Indistinguibilidad en nivel low de los tipos con nivel de seguridad y los tipos de la pila de control

$$\boxed{\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 \ \mathsf{wvalEq}}$$

El juicio anterior se cumple si:

- $w_1 = w_2$, o
- $w_1 = L[\overline{\Sigma_1}]$, $w_2 = L[\overline{\Sigma_2}]$, y existen las sustituciones γ_1 y γ_2, y una secuencia de tipos de pila $\overline{\Sigma}$ tal que:
  1. $Dom(\gamma_1) = Dom(\gamma_2) = FV(\overline{\Sigma})$
  2. $\overline{\Sigma_1} = \overline{\Sigma}\{\gamma_1\}$ y $\overline{\Sigma_2} = \overline{\Sigma}\{\gamma_2\}$, y
  3. para cada Σ_i en la secuencia $\overline{\Sigma}$, $\rhd_{\Psi_1,\Psi_2} \Sigma_i\{\gamma_1\} \approx_\zeta \Sigma_i\{\gamma_2\}$ $\mathsf{stackTypeEq}$.

$$\boxed{\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \sigma_1 \wedge \sigma_2 \ \mathsf{wval}}$$

$$\frac{l_1 \not\sqsubseteq \zeta \quad l_2 \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : {\tau_1}^{l_1} \wedge {\tau_2}^{l_2} \ \mathsf{wval}} \quad [\mathsf{EqWvalH}]$$

$$\frac{l_1 = l_2 \sqsubseteq \zeta \quad \rhd_{\Psi_1,\Psi_2} {\tau_1}^{l_1} \approx_\zeta {\tau_2}^{l_2} \ \mathsf{secTypeEq} \quad \rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 \ \mathsf{wvalEq}}{\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : {\tau_1}^{l_1} \wedge {\tau_2}^{l_2} \ \mathsf{wval}} \quad [\mathsf{EqWvalL}]$$

Figura 2.18: Indistinguibilidad de los valores

$$\boxed{\rhd_{\Psi_1,\Psi_2} h_1 \approx_\zeta h_2 : \sigma_1 \wedge \sigma_2 \text{ hval}}$$

$$\frac{l_1 \not\sqsubseteq \zeta \quad l_2 \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} h_1 \approx_\zeta h_2 : {\tau_1}^{l_1} \wedge {\tau_2}^{l_2} \text{ hval}} \quad \text{[EqHvalH]}$$

$$\frac{l_1 = l_2 \sqsubseteq \zeta \quad {\kappa_1}^{l_1}.B_1 = {\kappa_2}^{l_2}.B_2}{\rhd_{\Psi_1,\Psi_2} {\kappa_1}^{l_1}.B_1 \approx_\zeta {\kappa_2}^{l_2}.B_2 : {\kappa_1}^{l_1} \wedge {\kappa_2}^{l_2} \text{ hval}} \quad \text{[EqHvalBlkL]}$$

$$\frac{l_1 = l_2 \sqsubseteq \zeta \quad \rhd_{\Psi_1,\Psi_2} w_i \approx_\zeta w_i' : \sigma_i \wedge \sigma_i' \text{ wval} \;\text{ para todo } i \in \{1..n\}}{\rhd_{\Psi_1,\Psi_2} \langle w_1, \ldots, w_n\rangle \approx_\zeta \langle w_1', \ldots, w_n'\rangle : \langle \sigma_1, \ldots, \sigma_n\rangle^{l_1} \wedge \langle \sigma_1', \ldots, \sigma_n'\rangle^{l_2} \text{ hval}} \quad \text{[EqHvalTupL]}$$

$\boxed{\rhd H_1 \approx_\zeta H_2 : \Psi_1 \wedge \Psi_2 \text{ heap}}$ El juicio se cumple si y solo si $\ell \in Dom_\cup(\Psi_1, \Psi_2)$:

$$\texttt{label}(\Psi_1(\ell)) \sqsubseteq \zeta \text{ o } \texttt{label}(\Psi_2(\ell)) \sqsubseteq \zeta, \text{ implica } \begin{cases} \ell \in Dom_\cap(H_1, H_2, \Psi_1, \Psi_2), \text{ y} \\ \rhd_{\Psi_1,\Psi_2} H_1(\ell) \approx_\zeta H_2(\ell) : \Psi_1(\ell) \wedge \Psi_2(\ell) \text{ hval.} \end{cases}$$

$\boxed{\rhd_{\Psi_1,\Psi_2} R_1 \approx_\zeta R_2 : \Gamma_1 \wedge \Gamma_2 \text{ regBank}}$ El juicio se cumple si y solo si para todo $r \in Dom_\cup(\Gamma_1, \Gamma_2) \setminus \{\texttt{sp}\}$:

$$\texttt{label}(\Gamma_1(r)) \sqcup \texttt{label}(\Gamma_2(r)) \sqsubseteq \zeta, \text{ implica } \begin{cases} r \in Dom_\cap(R_1, R_2, \Gamma_1, \Gamma_2), \text{ y} \\ \rhd_{\Psi_1,\Psi_2} R_1(r) \approx_\zeta R_2(r) : \Gamma_1(r) \wedge \Gamma_2(r) \text{ wval.} \end{cases}$$

$\boxed{\rhd_{\Psi_1,\Psi_2} B_1 \approx_\zeta B_2 \text{ code}}$ se cumple si y solo si $\exists \gamma_1, \gamma_2, B$tal que :

1. $Dom(\gamma_1) = Dom(\gamma_2) = FV(B)$
2. $B_1 = B\{\gamma_1\}$ y $B_2 = B\{\gamma_2\}$ y
3. para todo $X \in Dom(\gamma_1)$, $\rhd_{\Psi_1,\Psi_2} X\{\gamma_1\} \approx_\zeta X\{\gamma_2\}$ stackTypeEq.

Figura 2.19: Indistinguibilidad de los valores del heap, heap, registros y bloques de código

Teorema 3 (no-interferencia). Dadas las siguientes configuraciones de máquina $\Pi_i = (H_i, R_i, B)$ y sus tipos $\Omega_i = [\Psi_i, \Gamma_i, \bot]$ para $i \in \{1, 2\}$. Si las siguientes condiciones se cumplen:

- $\Gamma_1(\mathtt{sp}) = \mathit{Halt} \cdot \Sigma_1$ y $\Gamma_2(\mathtt{sp}) = \mathit{Halt} \cdot \Sigma_2$
- $\triangleright\Pi_1 \approx_\zeta \Pi_2 : [\Psi_1, \Gamma_1, \bot] \wedge [\Psi_2, \Gamma_2, \bot]$ machConfig
- $\Pi_1' = (H_1', R_1', \mathtt{halt})$ y $\Pi_1 \twoheadrightarrow \Pi_1'$
- $\Pi_2' = (H_2', R_2', \mathtt{halt})$ y $\Pi_2 \twoheadrightarrow \Pi_2'$

Entonces existen los tipos $[\Psi_1', \Gamma_1', \mathbf{pc}_1']$ y $[\Psi_2', \Gamma_2', \mathbf{pc}_2']$ tal que:

$$\triangleright\Pi_1' \approx_\zeta \Pi_2' : [\Psi_1', \Gamma_1', \mathbf{pc}_1'] \wedge [\Psi_2', \Gamma_2', \mathbf{pc}_2'] \text{ machConfig.}$$

Para poder demostrar el teorema anterior tenemos que considerar dos tipos de reducciones, cuando el **pc** es *low* y cuando el **pc** es *high*. Para esto necesitamos los lemas: Reducción con **pc** Low y Reducción con **pc** High, presentados a continuación.

Lema 1 (Reducción con pc Low). Dadas las configuraciones de máquina $\Pi_i = (H_i, R_i, B_i)$ y sus tipos $\Omega_i = [\Psi_i, \Gamma_i, \mathbf{pc}_i]$ para $i \in \{1, 2\}$, tal que

1. $\triangleright\Pi_1 \approx_\zeta \Pi_2 : \Omega_1 \wedge \Omega_2$ machConfig,
2. $\mathbf{pc}_1 \sqsubseteq \zeta$ y $\mathbf{pc}_2 \sqsubseteq \zeta$, y
3. $\Pi_1 \longrightarrow \Pi_1'$.

Entonces existe una configuración de máquina Π_2' y los tipos $\Omega_1' = [\Psi_1', \Gamma_1', \mathbf{pc}_1']$ y $\Omega_2' = [\Psi_2', \Gamma_2', \mathbf{pc}_2']$ tal que

1. $\Pi_2 \longrightarrow \Pi_2'$ y
2. $\triangleright\Pi_1' \approx_\zeta \Pi_2' : [\Psi_1', \Gamma_1', \mathbf{pc}_1'] \wedge [\Psi_2', \Gamma_2', \mathbf{pc}_2']$ machConfig.

Esquema de prueba. El ítem $\Pi_2 \longrightarrow \Pi_2'$ concluye como consecuencia directa del Teorema 1 (progreso) dado que la primer y tercer hipótesis garantizan que $B_2 \neq \mathtt{halt}$. Para el ítem $\triangleright\Pi_1' \approx_\zeta \Pi_2' : [\Psi_1', \Gamma_1', \mathbf{pc}_1'] \wedge [\Psi_2', \Gamma_2', \mathbf{pc}_2']$ machConfig, por el Teorema 2 (preservación) sabemos que $\triangleright\Pi_1' : \Omega_1'$ machConfig y $\triangleright\Pi_2' : \Omega_2'$ machConfig. Luego, la demostración es por análisis de casos sobre la instrucción ejecutada en $\Pi_1 \longrightarrow \Pi_1'$.

□

Lema 2 (Reducción con pc High). Dadas las configuraciones de máquina $\Pi_i = (H_i, R_i, B_i)$ y sus tipos $\Omega_i = [\Psi_i, \Gamma_i, \mathbf{pc}_i]$ para $i \in \{1, 2\}$, tal que

1. $\triangleright\Pi_1 \approx_\zeta \Pi_2 : \Omega_1 \wedge \Omega_2$ machConfig,

2. $\mathbf{pc}_1 \not\sqsubseteq \zeta$ y $\mathbf{pc}_2 \not\sqsubseteq \zeta$, y
3. $\Pi_1 \longrightarrow \Pi_1'$.

Entonces:

- Π_2 diverge, o
- existe una configuración de máquina Π_2' y los tipos $\Omega_1' = [\Psi_1', \Gamma_1', \mathbf{pc}_1']$ y $\Omega_2' : [\Psi_2', \Gamma_2', \mathbf{pc}_2']$ tal que:
  1. $\Pi_2 \twoheadrightarrow \Pi_2'$ y
  2. $\triangleright\Pi_1' \approx_\zeta \Pi_2' : [\Psi_1', \Gamma_1', \mathbf{pc}_1'] \wedge [\Psi_2', \Gamma_2', \mathbf{pc}_2']$ machConfig.

El caso que presenta dificultades para la demostración de este lema es cuando el paso en la ejecución $\Pi_1 \longrightarrow \Pi_1'$ baja el nivel del **pc** al saltar a un *junction point* con nivel *low*. Aquí, es necesario encontrar una configuración de máquina Π_2' para la cual Π_1' y Π_2' sean indistinguibles-ζ. El principal problema es como garantizar que las pilas de control de Π_1 y Π_2 que previamente eran indistinguibles en nivel *high* y posiblemente de diferente tamaño, ahora pasen a ser de igual tamaño e indistinguibles en nivel *low*. Dado que la ejecución comenzó con ambos **pc**s en nivel *low*, sabemos que las pilas de Π_1 y Π_2 tienen en común una sub-pila indistinguible en nivel *low*. Tenemos que asegurarnos que esta sub-pila se vuelva la pila de control actual al alcanzar el bloque de código que representa al *junction point*. Esto es posible dado que los junction points son parte de la pila de control. Cuando en la reducción $\Pi_1 \longrightarrow \Pi_1'$ se salta a un junction point L, la pila de control de Π_1 tiene la forma $L \cdot \Sigma_1$. Además, el hecho de que el **pc** en $\Psi(L)$ es *low* y $\triangleright_{\Psi_1,\Psi_k} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge \Sigma_2$ cstackHigh, deducimos que $\Sigma_2 = ?_1 \cdot \ldots \cdot ?_n \cdot L \cdot \Sigma_2'$ y $S_2 = w_1 \cdot \ldots \cdot w_m \cdot S_2'$, $m \leq n$, donde los signos de pregunta "?" pueden ser junction points, tipos nonsense o tipos con nivel de seguridad. Por otro lado,

$$\triangleright_{\Psi_1,\Psi_k} S_1 \approx_\zeta S_2' : \Sigma_1 \wedge \Sigma_2' \text{ cstackLow} \tag{2.1}$$

se cumple por la definición del juicio cstackHigh. Podemos garantizar que los signos de pregunta en $\Sigma_2 = ?_1 \cdot \ldots \cdot ?_n \cdot L \cdot \Sigma_2'$ son tipos con nivel de seguridad *high*, por la siguiente definición y el siguiente lema:

Definición 2 (tope-ζ). Decimos que Σ es *tope-ζ* en Σ', si existen las etiquetas $L_1, \ldots, L_n$ (posiblemente ninguna) y los tipos de pila $\hat{\sigma}_{i,1}, \ldots, \hat{\sigma}_{i,k_i}$ para $i \in \{1..n\}$ (posiblemente ninguno) tal que:

- $\Sigma' = \hat{\sigma}_{1,1} \cdot \ldots \cdot \hat{\sigma}_{1,k_1} \cdot L_1 \cdot \hat{\sigma}_{2,1} \cdot \ldots \cdot \hat{\sigma}_{2,k_2} \cdot L_2 \cdot \ldots \cdot \hat{\sigma}_{n,1} \cdot \ldots \cdot \hat{\sigma}_{n,k_n} \cdot L_n \cdot \Sigma$
- $\Psi(L_i) = \texttt{CODE}\langle\forall[X_i]\Gamma_i \mid \mathbf{pc}_i\rangle^{l_i}$ implica que $\mathbf{pc}_i \sqcup l_i \not\sqsubseteq \zeta$, para todo $1 \leq i \leq n$.
- $\texttt{label}(\hat{\sigma}_{ij}) \not\sqsubseteq \zeta$, para todo $1 \leq i \leq n$ y $1 \leq j \leq k_i$.

Lema 3 (Poda por Izquierda). Si

- $\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge \Sigma_2$ cstackHigh o $\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge \Sigma_2$ cstackLow, y
- $\Psi_1(L) = \texttt{CODE}\langle \forall[\Theta]\Gamma \mid \mathbf{pc} \rangle^l$ con $\mathbf{pc} \sqcup l \not\sqsubseteq \zeta$

entonces $\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2$ cstackHigh.

Sabiendo que la ejecución que comienza con Π_2 termina, en algún momento el bloque de código L que representa el junction point será alcanzado. En este momento, las configuraciones de máquina se "sincronizan" de acuerdo a (2.1).

Esquema de prueba. La demostración del **Lema 2 (Reducción con pc High)** es por análisis de casos sobre la definición de $\Pi_1 \longrightarrow \Pi_1'$, utilizando el Lema Auxiliar definido a continuación, para el caso en el que el paso de ejecución sea el que alcanza el junction point que baja el **pc** a *low*. □

Esquema de prueba. La demostración del **Lema 3 (Poda por Izquierda)** es por inducción sobre la derivación de $\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge \Sigma_2$ cstackHigh y $\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge \Sigma_2$ cstackLow. □

Lema 4 (Auxiliar para Reducción con pc High). Dadas las configuraciones de máquina $\Pi_i = (H_i, R_i, B_i)$ y sus tipos $\Omega_i = [\Psi_i, \Gamma_i, \mathbf{pc}_i]$ para $i \in \{1..k\}$, tal que:

1. $\Pi_1 \twoheadrightarrow \Pi_k$,
2. $\rhd \Pi_i : \Omega_i$ machConfig, $i \in \{1..k\}$, y Ω_i que se obtiene del Teorema 2 (preservación), para $i \in \{2..k\}$.
3. $\mathbf{pc}_1 \not\sqsubseteq \zeta$, y
4. $\Gamma_k(\mathsf{sp}) = L \cdot \Sigma_k$, para algún L y Σ_k, y donde $\Gamma_i(\mathsf{sp})$ es tope-ζ para cada $i \in \{1..k\}$.

Entonces, los siguientes ítems se cumplen:

1. $\rhd H_1 \approx_\zeta H_k : \Psi_1 \wedge \Psi_k$ heap,
2. $\rhd_{\Psi_1,\Psi_k} R_1 \approx_\zeta R_k : \Gamma_1 \wedge \Gamma_k$ regBank,
3. $\rhd_{\Psi_1,\Psi_k} R_1(\mathsf{sp}) \approx_\zeta R_k(\mathsf{sp}) : \Gamma_1(\mathsf{sp}) \wedge \Gamma_k(\mathsf{sp})$ cstackHigh, y
4. $\mathbf{pc}_i \not\sqsubseteq \zeta$, para $i \in \{1..k\}$.

Esquema de prueba. La demostración del **Lema 4 (Auxiliar para Reducción con pc High)** es por inducción sobre la longitud de la derivación de $\Pi_1 \twoheadrightarrow \Pi_k$. □

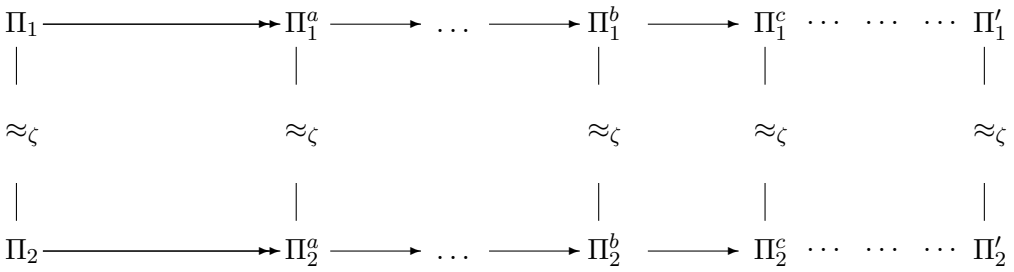

Figura 2.20: Esquema de Prueba de No-Interferencia

Esquema de prueba. Para la demostración del **Teorema 3 (no-interferencia)** tomemos dos ejecuciones de un programa Π como se muestra en la figura 2.20, donde:

- $\Pi_1 = (H_1, R_1, B) : [\Psi_1, \Gamma_1, \bot]$ y $\Pi_2 = (H_2, R_2, B) : [\Psi_2, \Gamma_2, \bot]$ con $\Gamma_1(\mathtt{sp}) = \mathit{Halt} \cdot \epsilon$ y $\Gamma_2(\mathtt{sp}) = \mathit{Halt} \cdot \epsilon$.
- $\Pi_1' = (H_1', R_1', \mathtt{halt})$ y $\Pi_2' = (H_2', R_2', \mathtt{halt})$.
- $\triangleright\Pi_1 \approx_\zeta \Pi_2 : [\Psi_1, \Gamma_1, \bot] \wedge [\Psi_2, \Gamma_2, \bot]$ machConfig
- $B \neq \mathtt{halt}$ (Si $B = \mathtt{halt}$ concluimos trivialmente.)

  Podemos comenzar aplicando el Lema 1 (Reducción con **pc** Low) dado que $\triangleright\Pi_1 \approx_\zeta \Pi_2 : \Omega_1 \wedge \Omega_2$ machConfig, $\bot \sqsubseteq \zeta$ y por $B \neq \mathtt{halt}$ y el Teorema 1 (progreso) tenemos $\Pi_1 \longrightarrow \Pi_1'$. Aplicamos repetidamente este lema hasta que no sea posible, obteniendo las configuraciones de máquina Π_1^a y Π_2^a para cada ejecución (note que en la figura 2.20 la flecha continua de doble cabeza significa que se realizaron varios pasos, sobre las mismas instrucciones para ambas ejecuciones). Luego, al no poder seguir aplicando el Lema 1 (Reducción con **pc** Low), tenemos que considerar los siguientes dos casos:

  - Si $\Pi_1^a = \Pi_1' = (H_1', R_1', \mathtt{halt})$ y $\Pi_2^a = \Pi_2' = (H_2', R_2', \mathtt{halt})$ entonces terminamos la ejecución y por Lema 1 (Reducción con **pc** Low) podemos decir que $\triangleright\Pi_1' \approx_\zeta \Pi_2' : [\Psi_1', \Gamma_1', \mathbf{pc}_1'] \wedge [\Psi_2', \Gamma_2', \mathbf{pc}_2']$ machConfig.
  - Si $B_1^a \neq B_2^a \neq \mathtt{halt}$ entonces por Lema 1 (Reducción con **pc** Low) sabemos que $\mathbf{pc}_1 \not\sqsubseteq \zeta$, $\mathbf{pc}_2 \not\sqsubseteq \zeta$ y $\Pi_1^a \approx_\zeta \Pi_2^a$. A partir de aquí ambas ejecuciones siguen independientemente (note las flechas en la figura 2.20, separadas por puntos suspensivos para indicar pasos independientes). Podemos aplicar repetidamente el Lema 2 (Reducción con **pc** High) hasta que no sea posible debido a que ejecutamos la directiva de tipado `jmpJP` que baja el nivel del **pc** a *low*. Asumiendo que obtenemos las configuraciones de máquina Π_1^c y Π_2^c, por Lema 2 (Reducción con **pc** High) podemos decir que $\Pi_1^c \approx_\zeta \Pi_2^c$. Aquí, una vez mas con el **pc** en nivel *low*, comenzamos nuevamente hasta alcanzar el final de la ejecución.

□

Como se mencionó anteriormente, para la demostración del Lema 2 (Reducción con **pc** High) se requiere la utilización del Lema Auxiliar para el caso en el que el paso a ejecutar sea el que alcanza al junction point que baja el pc a *low*. Note que en la figura 2.20 tenemos la configuración de máquina $\Pi_1^b = (H_1^b, R_1^b, B)$ donde $\Gamma_1^b(sp) = L.\Sigma_1^b$ y $B = \texttt{jmpJP}\ L[\Sigma]$. Luego, podemos encontrar la configuración de máquina $\Pi_2^b = (H_2^b, R_2^b, B)$ donde $\Gamma_2^b(sp) = L.\Sigma_2^b$ y $B = \texttt{jmpJP}\ L[\Sigma']$ tal que $\Pi_2^a \overset{k}{\twoheadrightarrow} \Pi_2^b$ y donde $L.\Sigma_2^b$ es tope-ζ para cada una de las pilas de control en las configuraciones de máquina intermedias para la secuencia de ejecuciones $\overset{k}{\twoheadrightarrow}$. Luego podemos aplicar el Lema Auxiliar para obtener $\triangleright H_2^a \approx_\zeta H_2^b : \Psi_2^a \wedge \Psi_2^b\ \mathsf{heap}$, $\triangleright_{\Psi_2^a,\Psi_2^b} R_2^a \approx_\zeta R_2^b : \Gamma_2^a \wedge \Gamma_2^b\ \mathsf{regBank}$, $\triangleright_{\Psi_2^a,\Psi_2^b} R_2^a(\mathtt{sp}) \approx_\zeta R_2^b(\mathtt{sp}) : \Gamma_2^a(\mathtt{sp}) \wedge \Gamma_2^b(\mathtt{sp})\ \mathsf{cstackHigh}$, y $\mathbf{pc}_i \not\sqsubseteq \zeta$, para $i \in \{1..k\}$. Y dado que $\Pi_1^a \approx_\zeta \Pi_2^a$ obtenemos $\Pi_1^b \approx_\zeta \Pi_2^b$. Luego podemos realizar el paso que alcanza el junction point y baja el nivel de seguridad del **pc**, obteniendo pilas indistinguibles en nivel *low*.

Capítulo 3

Función de Compilación

Como se menciono en el capítulo anterior para el caso de lenguajes de bajo nivel como SecTAL, es importante poder relacionar los resultados de no-interferencia con aquellos correspondientes a un lenguaje de alto nivel. En esta dirección, presentamos una *función de compilación* donde el lenguaje de origen es un lenguaje imperativo sencillo de alto nivel y el lenguaje destino es SecTAL. Luego mostramos que la compilación preserva tipos en el sentido de que genera programas SecTAL bien tipados. Nuestro lenguaje de alto nivel, al que llamaremos While, similar a aquel presentado en [SM03], está equipado con un sistema de tipos que nos permite comprobar que los programas bien tipados cumplen con la propiedad de no-interferencia. De esta forma el lenguaje origen de nuestra función de compilación cumple con la propiedad de no-interferencia al igual que nuestro lenguaje de destino, SecTAL.

Comenzaremos definiendo While. La sintaxis, la semántica operacional y su sistema de tipos con niveles de seguridad. Luego expresamos la noción de no-interferencia de forma que nos permita mostrar que los programas bien tipados cumplen con esta propiedad. Finalmente, se presenta la función de compilación y la demostración de que preserva de tipos.

3.1. El Lenguaje de Alto Nivel While

3.1.1. Sintaxis

Comenzaremos presentando las diferentes categorías sintácticas del lenguaje While y las meta variables utilizadas para denotar diferentes construcciones sintácticas. Luego definimos la sintaxis en notación similar a BNF. Las categorías sintácticas de While son:

- e representa las expresiones aritméticas. Utilizaremos $e_1, e_2, \ldots, e_n$ para denotar expresiones aritméticas.

- be representa las expresiones booleanas. Utilizaremos $be_1, be_2, \ldots, be_n$ para denotar expresiones booleanas.
- i constantes enteras. Los números enteros, junto con las constantes *true* y *false*, son los únicos valores del lenguaje. Una expresión e reduce a un valor entero y una expresión be reduce a un valor *true* o *false*. Utilizaremos $v_1, v_2, \ldots, v_n$ para denotar valores, **Val** como un conjunto de valores enteros y **BooleanVal** como un conjunto de valores booleanos.
- x variables. Los nombres de las variables pueden ser cadenas de caracteres y dígitos comenzando por un caracter. Utilizaremos $x_1, x_2, \ldots, x_n$ para denotar variables. **Var** como un conjunto de variables.
- Las sentencias representadas por stm pueden ser, asignación, sentencias de condición, sentencias de repetición, y sentencias de declaración de variables. Toda variable debe ser declarada e inicializada antes de ser utilizada.
- Finalmente, t que puede ser L o H representa nivel de seguridad público y nivel de seguridad privado respectivamente.

A continuación, definimos la sintaxis del lenguaje en notación similar a BNF:

$$
\begin{array}{rll}
e & ::= i \mid x \mid e_1 \ + \ e_2 & \text{(expresiones aritméticas)} \\
be & ::= \mathit{true} \mid \mathit{false} \mid e_1 \ = \ 0 & \text{(expresiones booleanas)} \\
stm & ::= \ x \ := \ e & \text{(sentencias)} \\
 & \quad \mid \ stm_1 \ ; \ stm_2 & \\
 & \quad \mid \ \textbf{if } be \textbf{ then } stm \textbf{ else } stm \textbf{ end} & \\
 & \quad \mid \ \textbf{while } be \textbf{ do } stm \textbf{ end} & \\
 & \quad \mid \ \textbf{var } x : t \ := \ e \textbf{ in } stm \textbf{ end} & \\
prog & ::= stm & \text{(programas)} \\
t & ::= \mathsf{L} \mid \mathsf{H} & \text{(tipos)}
\end{array}
$$

3.1.2. Semántica Operacional

La semántica operacional de WHILE la definiremos a partir de *funciones semánticas*. Estas funciones toman como entrada un elemento sintáctico y devuelven su significado. Para comenzar necesitamos definir la noción de *estado*.

Una sentencia stm es ejecutada bajo un estado σ, quien asocia valores a variables. A su vez, el significado de una expresión depende del valor ligado a la o las variables dentro de la expresión. Representamos un estado como una función que mapea variables

a valores:

$$\textbf{Estado} = \textbf{Var} \rightarrow \textbf{Val}$$

por ejemplo, $\sigma = \{x_1 \mapsto v_1, x_2 \mapsto v_2, x_3 \mapsto v_3, \ldots, x_n \mapsto v_n\}$, donde $x_1, x_2, x_3, \ldots, x_n$ son elementos del conjunto **Var** y $v_1, v_2, v_3, \ldots, v_n$ son elementos del conjunto **Val**.

Habiendo definido la noción de estado, dada una expresión e y un estado σ, podemos determinar su valor. Para ello definimos una función $\mathfrak{A}$ que recibe dos parámetros: la construcción sintáctica de una expresión y un estado.

$$\mathfrak{A} : e \rightarrow (\textbf{Estado} \rightarrow \textbf{Val})$$

$\mathfrak{A}$ mapea una expresión aritmética e y un estado σ a un valor $\mathfrak{A}[e]\sigma$. A continuación definimos $\mathfrak{A}$:

$$\begin{aligned}
\mathfrak{A}[i]\sigma &= i \\
\mathfrak{A}[x]\sigma &= \sigma(x) \\
\mathfrak{A}[e_1 + e_2]\sigma &= \mathfrak{A}[e_1]\sigma + \mathfrak{A}[e_2]\sigma
\end{aligned}$$

De igual forma procedemos con las expresiones booleanas. Definimos una función $\mathfrak{B}$ que recibe dos parámetros: la construcción sintáctica de una expresión booleana y un estado.

$$\mathfrak{B} : be \rightarrow (\textbf{Estado} \rightarrow \textbf{BooleanVal})$$

$\mathfrak{B}$ mapea una expresión booleana be y un estado σ a un valor $\mathfrak{B}[be]\sigma$. Definimos $\mathfrak{B}$ de la siguiente forma:

$$\begin{aligned}
\mathfrak{B}[true]\sigma &= true \\
\mathfrak{B}[false]\sigma &= false \\
\mathfrak{B}[e_1 = 0]\sigma &= \begin{cases} true & si\ \mathfrak{A}[e_1]\sigma = 0 \\ false & en\ otro\ caso \end{cases}
\end{aligned}$$

Vamos a expresar la ejecución de sentencias *stm* utilizando semántica operacional del tipo big-step [GK87]. Definimos una relación de transición $\Downarrow$ sobre configuraciones, donde una configuración, puede ser un par $\langle stm, \sigma \rangle$ o simplemente σ. El primero representa que la sentencia *stm* va a ser ejecutada en el estado σ, y el segundo representa un estado final. De esta forma, las transiciones tienen la siguiente forma:

- $\langle stm, \sigma \rangle \Downarrow \sigma'$: la ejecución de *stm* en σ ha terminado y el estado final es σ'.

Las reglas y axiomas de transición que definen $\Downarrow$ se presentan en la Figura 3.1. Para indicar modificaciones en un estado σ utilizamos la siguiente notación: $\sigma[y \mapsto v]$. Esto significa que si ya existía una asociación en σ para y, ahora se sobrescribe por $y \mapsto v$, si no existía, se agrega una nueva asociación a σ.

El sistema de transición definido en la Figura 3.1 es determinístico. Podemos definir el siguiente lema:

$$\langle x := e, \sigma\rangle \Downarrow \sigma[x \mapsto \mathfrak{A}[e]\sigma] \qquad [\mathsf{E}-\mathsf{Asig}]$$

$$\frac{\langle stm_1, \sigma\rangle \Downarrow \sigma', \langle stm_2, \sigma'\rangle \Downarrow \sigma''}{\langle stm_1; stm_2, \sigma\rangle \Downarrow \sigma''} \qquad [\mathsf{E}-\mathsf{Stm}]$$

$$\frac{\langle stm_1, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } be \textbf{ then } stm_1 \textbf{ else } stm_2 \textbf{ end}, \sigma\rangle \Downarrow \sigma'} \; if \;\; \mathfrak{B}[be]\sigma \;=\; true \qquad [\mathsf{E}-\mathsf{IFTrue}]$$

$$\frac{\langle stm_2, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } be \textbf{ then } stm_1 \textbf{ else } stm_2 \textbf{ end}, \sigma\rangle \Downarrow \sigma'} \; if \;\; \mathfrak{B}[be]\sigma \;=\; false \qquad [\mathsf{E}-\mathsf{IFFalse}]$$

$$\frac{\langle stm, \sigma\rangle \Downarrow \sigma', \langle \textbf{while } be \textbf{ do } stm \textbf{ end}, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while } be \textbf{ do } stm \textbf{ end}, \sigma\rangle \Downarrow \sigma''} \; if \;\; \mathfrak{B}[be]\sigma \;=\; true \qquad [\mathsf{E}-\mathsf{WHILETrue}]$$

$$\langle \textbf{while } be \textbf{ do } stm \textbf{ end}, \sigma\rangle \Downarrow \sigma \; if \;\; \mathfrak{B}[be]\sigma \;=\; false \qquad [\mathsf{E}-\mathsf{WHILEFalse}]$$

$$\frac{\langle stm, \sigma[x \mapsto \mathfrak{A}[e]\sigma]\rangle \Downarrow \sigma'}{\langle \textbf{var } x : t \;:=\; e \textbf{ in } stm \textbf{ end}, \sigma\rangle \Downarrow \sigma'[x \mapsto \sigma(x)]} \qquad [\mathsf{E}-\mathsf{Decl}]$$

Figura 3.1: Semántica operacional de WHILE

Lema 1 (semántica determinística). Para cualquier stm, σ, σ' y σ'' tenemos que $\langle stm, \sigma\rangle \Downarrow \sigma'$ y $\langle stm, \sigma\rangle \Downarrow \sigma''$ entonces $\sigma' = \sigma''$.

La semántica de las sentencias puede expresarse como una función que va de un estado a otro estado.

$$\mathfrak{S} : Stm \to (\textbf{Estado} \to \textbf{Estado})$$

$$\mathfrak{S}[stm]\sigma = \begin{cases} \sigma' & si\ \langle stm, \sigma\rangle \Downarrow \sigma' \\ indefinido & en\ otro\ caso \end{cases}$$

3.1.3. Sistema de Tipos

A continuación describiremos el sistema de tipos con niveles de seguridad para WHILE, el cual será utilizado para garantizar la propiedad de no-interferencia. Tenemos que definir el sistema de tipos de modo que podamos detectar flujos de información inválidos, sean explícitos o implícitos. Para ello clasificaremos las expresiones con nivel de seguridad L (público) o H (privado). Las sentencias se ejecutarán bajo un contexto de seguridad que también puede ser L o H. En nuestro caso el contexto de seguridad es simplemente la etiqueta *pc*. Como se mencionó anteriormente se utilizará la etiqueta *pc* para detectar flujos de información implícitos.

Un contexto de tipado es una función Δ, que mapea variables a sus niveles de seguridad o tipos. $\Delta = \mathbf{Var} \rightarrow \mathbf{Tipo}$, $\mathbf{Tipo} = \{\mathsf{L}, \mathsf{H}\}$. Por ejemplo, $\Delta = \{x_1 \mapsto t_1, x_2 \mapsto t_2, x_3 \mapsto t_3, \ldots, x_n \mapsto t_n\}$, donde $x_1, x_2, x_3, \ldots, x_n$ son elementos del conjunto **Var** y $t_1, t_2, t_3, \ldots, t_n$ son elementos del conjunto **Tipo**. Δ es el ambiente de declaración. Luego, escribimos $\Delta \vdash e : t$ para decir que e es tipable bajo el ambiente Δ y tiene tipo t o lo que es lo mismo en nuestro caso, nivel de seguridad t. Y escribimos $\Delta, pc \vdash stm$ para decir que la sentencia *stm* es tipable bajo el ambiente Δ y el contexto de seguridad *pc*.

En la Figura 3.2 y 3.3 se muestran las reglas y axiomas de tipado para las expresiones y sentencias de WHILE. Comenzando por las expresiones, podemos notar que las constantes enteras y booleanas son tratadas como públicas, dado que tendrán tipo L. Las variables serán tipadas con tipo t si se encuentran en el ambiente de declaración con tipo t. Finalmente, la comparación tendrá tipo t, si sus componentes son de tipo t y la suma tendrá el tipo que salga de la *junta* entre los tipos de sus componentes. Definimos la *junta* entre t_1 y t_2 de la siguiente forma:

$$t_1 \sqcup t_2 = \begin{cases} \mathsf{L} & si\ t_1 = \mathsf{L}\ y\ t_2 = \mathsf{L} \\ \mathsf{H} & en\ cualquier\ otro\ caso \end{cases}$$

Siguiendo con las sentencias podemos ver que es seguro tipar una sentencia de asignación siempre que la variable del lado izquierdo sea H. Para que sea seguro tipar una sentencia de asignación donde la variable del lado izquierdo es L, el contexto de seguridad *pc* debe ser L y la expresión del lado derecho debe ser L también. Las reglas $[\mathsf{T} - \mathsf{IF}]$ y $[\mathsf{T} - \mathsf{WHILE}]$ dicen que si la expresión booleana de la condición tiene nivel de seguridad H, entonces las sentencias tanto del cuerpo del **while** como de las ramas del **if** deben tiparse con contexto de seguridad H. Aquí se puede notar que la regla $[\mathsf{T} - \mathsf{AsigLow}]$ en combinación con las demás nos sirven para rechazar programas como inseguros cuando poseen flujos de información implícitos. La regla $[\mathsf{T} - \mathsf{Sub}]$ dice que si un programa es tipable en contexto de seguridad H también lo será en contexto de seguridad L.

Sobre las reglas de declaración de variables, no importa en que contexto de seguridad estemos, siempre que declaremos una variable con nivel de seguridad H. Si el contexto

$$\Delta \vdash \mathit{true} : \mathsf{L} \qquad [\mathsf{T} - \mathsf{BETrue}]$$

$$\Delta \vdash \mathit{false} : \mathsf{L} \qquad [\mathsf{T} - \mathsf{BEFalse}]$$

$$\Delta \vdash i : \mathsf{L} \qquad [\mathsf{T} - \mathsf{EConst}]$$

$$\frac{\Delta(x) = t}{\Delta \vdash x : t} \qquad [\mathsf{T} - \mathsf{EVar}]$$

$$\frac{\Delta \vdash e_1 : t}{\Delta \vdash e_1 = 0 : t} \qquad [\mathsf{T} - \mathsf{BEIgual}]$$

$$\frac{\Delta \vdash e_1 : t_1 \quad \Delta \vdash e_2 : t_2}{\Delta \vdash e_1 + e_2 : t_1 \sqcup t_2} \qquad [\mathsf{T} - \mathsf{ESuma}]$$

Figura 3.2: Reglas y axiomas de tipado para las expresiones de WHILE

de seguridad es L, entonces la variable a declarar debe ser L. Finalmente, la regla [T − Prog] intentará tipar un programa con contexto de seguridad L. Si bien el sistema de tipos no necesita de esta regla para detectar flujos inválidos, es necesaria como punto de entrada para la función de compilación y así poder inicializar la máquina abstracta de SECTAL.

La Figura 3.4 muestra algunos ejemplos de programas en WHILE que pasan el chequeo de tipos y otros que son rechazados como inválidos. El programa **(a)** pasa el chequeo de tipos. Note que requiere primero utilizar la regla [T − Sub] antes de poder utilizar la regla [T − Stm] para poder tipar la sección de código:

```
if a = 0 then a := 10 else b := 2 end;
if b = 0 then b := 1  else b := 2 end
```

El programa **(b)** es rechazado ya que posee un flujo explícito inválido. El programa **(c)** también es rechazando ya que contiene un flujo implícito inválido.

3.1.4. No-Interferencia

Como mencionamos anteriormente, un programa escrito en WHILE que pasa el chequeo de tipos cumple con la propiedad de no-interferencia. Cumplir con esta propiedad, intuitivamente significa que dado un programa *prog* bien tipado, las variables de nivel

$$\frac{\Delta(x) = \mathsf{H}}{\Delta, pc \vdash x := e} \quad [\mathsf{T} - \mathsf{AsigHigh}]$$

$$\frac{\Delta \vdash e : \mathsf{L} \quad \Delta(x) = \mathsf{L}}{\Delta, \mathsf{L} \vdash x := e} \quad [\mathsf{T} - \mathsf{AsigLow}]$$

$$\frac{\Delta \vdash be : pc \quad \Delta, pc \vdash stm_1 \quad \Delta, pc \vdash stm_2}{\Delta, pc \vdash \textbf{if } be \textbf{ then } stm_1 \textbf{ else } stm_2 \textbf{ end}} \quad [\mathsf{T} - \mathsf{IF}]$$

$$\frac{\Delta \vdash be : pc \quad \Delta, pc \vdash stm}{\Delta, pc \vdash \textbf{while } be \textbf{ do } stm \textbf{ end}} \quad [\mathsf{T} - \mathsf{WHILE}]$$

$$\frac{\Delta, pc \vdash stm_1 \quad \Delta, pc \vdash stm_2}{\Delta, pc \vdash stm_1; stm_2} \quad [\mathsf{T} - \mathsf{Stm}]$$

$$\frac{\Delta, \mathsf{H} \vdash stm}{\Delta, \mathsf{L} \vdash stm} \quad [\mathsf{T} - \mathsf{Sub}]$$

$$\frac{\Delta[x \mapsto \mathsf{H}], pc \vdash stm}{\Delta, pc \vdash \textbf{var } x \ : \ \mathsf{H} \ := \ e \textbf{ in } stm \textbf{ end}} \quad [\mathsf{T} - \mathsf{DecHigh}]$$

$$\frac{\Delta \vdash e : \mathsf{L} \quad \Delta[x \mapsto \mathsf{L}], \mathsf{L} \vdash stm}{\Delta, \mathsf{L} \vdash \textbf{var } x \ : \ \mathsf{L} \ := \ e \textbf{ in } stm \textbf{ end}} \quad [\mathsf{T} - \mathsf{DecLow}]$$

$$\frac{\Delta, \mathsf{L} \vdash stm}{\Delta \vdash stm} \quad [\mathsf{T} - \mathsf{Prog}]$$

Figura 3.3: Reglas y axiomas de tipado para las sentencias de While

```
(a) var a: L := 1 in
        var b: H := 0 in
            if a = 0 then a := 10 else b := 2 end;
            if b = 0 then b := 1  else b := 2 end
        end
    end

(b) var a: L := 1 in
        var b: H := 0 in
            a := b;
        end
    end

(c) var a: L := 1 in
        var b: H := 0 in
            if b = 0 then b := 1 else a := 2 end
        end
    end
```

Figura 3.4: Ejemplos de programas escritos en WHILE

de seguridad H no afectarán a las variables de nivel de seguridad L. Es decir que dada una variable x con nivel de seguridad L, uno podría cambiar los valores de entrada, para un programa, de las variables con nivel de seguridad H, ejecutar el programa y obtener que la variable x mantuvo su valor (considerando que el programa termina su ejecución).

Para establecer que nuestro sistema de tipos garantiza no-interferencia, primero vamos a definir la relación de equivalencia $\equiv_L^\Delta$ que describe que dos estados comparten los mismos valores para todas sus variables con nivel de seguridad L.

Definición 1 (equivalencia de estados). Dados dos estados σ_1 y σ_2 y un contexto de tipado Δ, $\sigma_1 \equiv_L^\Delta \sigma_2 \Leftrightarrow \forall x \in \mathrm{dom}(\Delta), \Delta(x) = \mathsf{L} \Rightarrow \sigma_1(x) = \sigma_2(x)$.

Lema 2 (relación de equivalencia). La relación $\equiv_L^\Delta$ es de equivalencia.

Definición 2 (programas no-interferentes). Un programa escrito en WHILE es no-interferente, denotado así $\mathrm{NI}_\Delta(\mathrm{stm})$, si dados los estados $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ donde $\sigma_1 \equiv_L^\Delta \sigma_2$, si $\langle stm, \sigma_1\rangle \Downarrow \sigma_3$ y $\langle stm, \sigma_2\rangle \Downarrow \sigma_4$ implica $\sigma_3 \equiv_L^\Delta \sigma_4$.

Los programas escritos en WHILE que pasan el chequeo de tipos cumplen con no-interferencia:

Teorema 2 (no-interferencia). Dado un programa *stm* escrito en WHILE, si $\Delta \vdash stm$ implica $\mathrm{NI}_\Delta(\mathrm{stm})$.

La demostración del Teorema 2 puede consultarse en el Apéndice A.

3.2. Función de Compilación

En esta sección se presenta una función de compilación del lenguaje While a SecTAL. También se incluye la prueba de que la compilación genera programas SecTAL que pasan el chequeo de tipos. Utilizaremos la función τ que mapea tipos de While a SecTAL y la definimos de la siguiente forma: $\tau(\mathsf{L}) = int^{\bot}$ y $\tau(\mathsf{H}) = int^{\top}$. Para simplificar la lectura y escritura de la función de compilación y la prueba omitiremos el tipo de dato *int* de SecTAL, de esta forma la función τ quedaría: $\tau(\mathsf{L}) = \bot$ y $\tau(\mathsf{H}) = \top$. Asumimos que la traducción de un programa bien tipado $\Delta, pc \vdash prog$ empieza en el heap H_0 cuyo tipo es determinado por Ψ_0 y que satisface $\triangleright H_0 : \Psi_0$ heap.

Utilizaremos la función *offset* que dada una variable y la pila de SecTAL devuelve su desplazamiento. Esta función se calcula en tiempo de compilación. El lenguaje While obliga a declarar las variables, por lo tanto vamos a asumir que por cada variable que se encuentra en una expresión e o sentencia stm, existirá una entrada en la pila de SecTAL con el mismo tipo. De esta forma: $\tau(\Delta(x)) = \Gamma(sp)[\mathit{offset}(x, \Sigma)]$. Esta correspondencia la denotamos así: $\Delta \sim \Gamma(sp)$. Σ, Γ y sp son los definidos en el capítulo anterior.

3.2.1. Traducción de Expresiones

La función de traducción de expresiones $|e|$ tiene la siguiente forma:

$$
\begin{array}{rcl}
|x| & = & \texttt{sld}\ r_a,\ sp[\mathit{offset}(x, \Sigma)] \\
|i| & = & \texttt{mov}\ r_a,\ i \\
|e_1 + e_2| & = & |e_1| \\
 & & \texttt{salloc}\ 1; \\
 & & \texttt{sst}\ sp[0],\ r_a; \\
 & & |e_2| \\
 & & \texttt{sld}\ r_b,\ sp[0]; \\
 & & \texttt{add}\ r_a,\ r_b,\ r_a; \\
 & & \texttt{sfree}\ 1
\end{array}
$$

Tener en cuenta que cuando accedemos a la pila con índice cero, siempre accedemos al tope. La función de traducción de expresiones booleanas $|be|$ tiene la siguiente forma:

$$
\begin{array}{rcl}
|true| & = & \texttt{mov}\ r_a,\ 1 \\
|false| & = & \texttt{mov}\ r_a,\ 0 \\
|e_1 = 0| & = & |e_1|
\end{array}
$$

3.2.2. Traducción de Sentencias

La traducción de sentencias se realiza en función del árbol de derivación de tipos del lenguaje fuente While. De esta forma, cada regla de tipado tendrá su propia traducción. Las reglas de tipado cuya hipótesis requiera generar bloques de código SecTAL, como por ejemplo $[\mathsf{T}-\mathsf{IF}]$, se escribirán con una notación de la forma: $\frac{\mathsf{TH}}{\mathsf{TC}}$. Donde TH representa la traducción de la hipótesis de la regla en cuestión y TC la traducción de la conclusión de la regla en cuestión.

La función de traducción de sentencias tiene la siguiente forma:

$$\left|\frac{d}{\Delta, pc \vdash stm}\right| \quad \begin{bmatrix} \Psi \\ H \\ \Gamma, pc, l_{inicio}, l_{fin}, \Sigma \end{bmatrix} = \begin{bmatrix} \Psi' \\ H' \\ \Sigma \end{bmatrix}$$

Donde:

- d: Usamos d para representar posibles derivaciones de tipos.
- Ψ: contexto de tipado del heap.
- H: el heap, donde $\triangleright H : \Psi$. El heap lo utilizaremos para ir agregando los bloques de código a medida que son traducidos del lenguaje While al lenguaje SecTAL. Utilizaremos la notación $H\{l = \dots\}$ y $\Psi\{l = \dots\}$ para denotar que el bloque de código l se agrega a H y el tipo del bloque de código l se agrega a Ψ.
- Γ: Tipado de registros y pila.
- Σ: Tipos de la pila de control.
- pc: El pc que se utilizó para tipar stm.
- l_{inicio}: Etiqueta donde comienza la primera instrucción. l_{inicio} se agregará a Ψ y H.
- l_{fin}: Etiqueta a donde saltará la última instrucción de stm. La función de traducción asegurará que Ψ ya contenga el tipo de l_{fin}. No es necesario esto mismo para el bloque de código, dado que antes de la ejecución de un programa, habiendo éste pasado el chequeo de tipos, todos los bloques ya habrán sido generados.

La traducción de un programa While, cuya regla de tipado es [T-Prog], genera dos bloques de código e inicializa la pila de SecTAL con la etiqueta *Halt*. El bloque inicial llamado l_{main} solo realiza un salto a la primera instrucción que genera la traducción de stm. Luego, el bloque de código l_{halt} contiene la instrucción $halt$ que finaliza el

programa SecTAL. Note que el tipo de l_{halt} se agrega a Ψ, cuando Ψ es pasado por parámetro para la traducción de la hipótesis de la regla de tipado en cuestión. Esto nos permite cumplir con lo definido arriba sobre l_{fin} para los sub-componentes del árbol de derivación de tipos. El bloque de código que generará la traducción de *stm*, finalizará con la instrucción `jmp` $l_{halt}[X]$, donde el tipo de l_{halt} ya existirá en Ψ al momento de intentar tipar dicho bloque.

$$\left|\frac{d}{\Delta, \mathsf{L} \vdash stm}\right| \begin{bmatrix} \Psi\{l_{halt} = \texttt{CODE}\langle\forall[X]\{sp : Halt.X\} \mid \bot\rangle^{\bot}\} \\ H \\ \Gamma, \bot, l, l_{halt}, Halt.X \end{bmatrix} = \begin{bmatrix} \Psi' \\ H' \\ Halt.X \end{bmatrix}$$

$$\left|\frac{\Delta, \mathsf{L} \vdash stm}{\Delta \vdash stm}\right| \begin{bmatrix} \Psi \\ H \\ \Gamma, l_{main}, l_{halt}, Halt.X \end{bmatrix} = \begin{bmatrix} \Psi'\{l_{main} = \texttt{CODE}\langle\forall[X]\{sp : Halt.X\} \mid \bot\rangle^{\bot}\} \\ H'\{l_{main} = \texttt{CODE}\langle\forall[X]\{sp : Halt.X\} \mid \bot\rangle^{\bot} \\ \qquad\qquad \texttt{jmp}\ l[X], \\ \\ \qquad l_{halt} = \texttt{CODE}\langle\forall[X]\{sp : Halt.X\} \mid \bot\rangle^{\bot} \\ \qquad\qquad \texttt{halt}\} \\ X \end{bmatrix}$$

La traducción de la sentencia de asignación se muestra a continuación:

$$\left|\frac{\Delta(x) = \mathsf{H}}{\Delta, pc \vdash x\ :=\ e}\right| \begin{bmatrix} \Psi \\ H \\ \Gamma, pc, l, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^{\bot}\} \\ H\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^{\bot} \\ \qquad\qquad |e| \\ \qquad\qquad \texttt{sst}\ sp[\mathit{offset}(x, \Sigma)],\ r_a; \\ \qquad\qquad \texttt{jmp}\ l'[X]\} \\ \Sigma \end{bmatrix}$$

$$\left|\frac{\Delta \vdash e : \mathsf{L} \quad \Delta(x) = \mathsf{L}}{\Delta, \mathsf{L} \vdash x\ :=\ e}\right| \begin{bmatrix} \Psi \\ H \\ \Gamma, \mathsf{L}, l, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid \mathsf{L}\rangle^{\bot}\} \\ H\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid \mathsf{L}\rangle^{\bot} \\ \qquad\qquad |e| \\ \qquad\qquad \texttt{sst}\ sp[\mathit{offset}(x, \Sigma)],\ r_a; \\ \qquad\qquad \texttt{jmp}\ l'[X]\} \\ \Sigma \end{bmatrix}$$

La traducción de la sentencia de declaración que se muestra a continuación, se divide en dos partes. La primera parte asigna memoria en la pila para almacenar el resultado de la traducción de la expresión e. La posición alocada tiene como tipo de dato $\tau(\mathsf{H})$. Luego se realiza la traducción de la sentencia *stm* que comienza en la etiqueta l_1. Finalmente la ejecución continua en la etiqueta l_2 quien libera el tope de la pila y salta a l'.

$$\left|\frac{d}{\Delta[x \mapsto \mathsf{H}], pc \vdash stm}\right| \begin{bmatrix} \Psi\{l_2 = \texttt{CODE}\langle\forall[X]\{sp : \top.\Sigma\} \mid pc\rangle^{\bot}\} \\ H \\ \Gamma, pc, l_1, l_2, \top.\Sigma \end{bmatrix} = \begin{bmatrix} \Psi_1 \\ H_1 \\ \top.\Sigma \end{bmatrix}$$

$$\left|\frac{\Delta[x \mapsto \mathsf{H}], pc \vdash stm}{\Delta, pc \vdash var\ x\ :\ \mathsf{H}\ :=\ e\ in\ stm\ end}\right| \quad \begin{bmatrix} \Psi \\ H \\ \Gamma, pc, l, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi_1\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^{\bot}\} \\ H_1\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^{\bot} \\ \qquad\qquad |e| \\ \qquad\qquad \texttt{salloc}\ 1; \\ \qquad\qquad \texttt{sstNsl}\ \tau(\mathsf{H}), sp[0],\ r_a; \\ \qquad\qquad \texttt{jmp}\ l_1[X], \\ \\ \qquad l_2 = \texttt{CODE}\langle\forall[X]\{sp : \top.\Sigma\} \mid pc\rangle^{\bot} \\ \qquad\qquad \texttt{sfree}\ 1; \\ \qquad\qquad \texttt{jmp}\ l'[X]\} \\ \Sigma \end{bmatrix}$$

Note que la instrucción de SecTAL `sstNsl` utilizada arriba, guarda en la pila el registro r_a con tipo $\tau(\mathsf{H}) = \top$ independientemente del tipo actual de r_a (ver regla de tipado [T-SstNsl] en el capítulo 2). Necesitamos dicha instrucción dado que la traducción de la expresión $|e|$ podría ser $\tau(\mathsf{L})$ dejando al registro r_a con tipo $\bot$. De esta forma, como estamos traduciendo una declaración de una variable x con tipo H, nos aseguramos que la misma se guarde en la pila con su tipo equivalente, $\tau(\mathsf{H})$ respetando la definición dada al comienzo de la sección que dice que $\Delta \sim \Gamma(sp)$. Como mencionamos anteriormente, dicha instrucción luego de realizado el chequeo de tipos, puede ser reemplazada por la instrucción `sst`.

A continuación procedemos de la misma forma que antes con la diferencia que la posición asignada en la pila tiene como tipo de dato $\tau(\mathsf{L})$.

$$\left|\frac{d}{\Delta[x \mapsto \mathsf{L}], \mathsf{L} \vdash stm}\right| \begin{bmatrix} \Psi\{l_2 = \texttt{CODE}\langle\forall[X]\{sp : \bot.\Sigma\} \mid \bot\rangle^{\bot}\} \\ H \\ \Gamma, \bot, l_1, l_2, \bot.\Sigma \end{bmatrix} = \begin{bmatrix} \Psi_1 \\ H_1 \\ \bot.\Sigma \end{bmatrix}$$

$$\left|\frac{\Delta \vdash e : \mathsf{L} \quad \Delta[x \mapsto \mathsf{L}], \mathsf{L} \vdash stm}{\Delta, \mathsf{L} \vdash var\ x\ :\ \mathsf{L}\ :=\ e\ in\ stm\ end}\right| \quad \begin{bmatrix} \Psi \\ H \\ \Gamma, \bot, l, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi_1\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid \bot\rangle^{\bot}\} \\ H_1\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid \bot\rangle^{\bot} \\ \qquad\qquad |e| \\ \qquad\qquad \texttt{salloc}\ 1; \\ \qquad\qquad \texttt{sst}\ sp[0],\ r_a; \\ \qquad\qquad \texttt{jmp}\ l_1[X], \\ \\ \qquad l_2 = \texttt{CODE}\langle\forall[X]\{sp : \bot.\Sigma\} \mid \bot\rangle^{\bot} \\ \qquad\qquad \texttt{sfree}\ 1; \\ \qquad\qquad \texttt{jmp}\ l'[X]\} \\ \Sigma \end{bmatrix}$$

La traducción de la sentencia de selección comienza por la traducción de la expresión booleana, luego inserta el *junction point* l' y salta dependiendo de la condición a alguna de las ramas. Ambas ramas terminan saltando a la etiqueta l_{iJP} quien finalmente salta al *junction point* l'.

$$\left|\frac{d_1}{\Delta, pc \vdash stm_1}\right| \begin{bmatrix} \Psi\{l_{iJP} = \texttt{CODE}\langle\forall[X]\{sp : l'.\Sigma\} \mid pc\rangle^{\perp}\} \\ H \\ \Gamma, pc, l_{then}, l_{iJP}, l'.\Sigma \end{bmatrix} = \begin{bmatrix} \Psi_1 \\ H_1 \\ l'.\Sigma \end{bmatrix}$$

$$\left|\frac{d_2}{\Delta, pc \vdash stm_2}\right| \begin{bmatrix} \Psi_1 \\ H_1 \\ \Gamma, pc, l_{else}, l_{iJP}, l'.\Sigma \end{bmatrix} = \begin{bmatrix} \Psi_2 \\ H_2 \\ l'.\Sigma \end{bmatrix}$$

$$\left|\frac{\Delta \vdash be : pc \ \Delta, pc \vdash stm_1 \ \Delta, pc \vdash stm_2}{\Delta, pc \vdash if \ be \ then \ stm_1 \ else \ stm_2 \ end}\right| \begin{bmatrix} \Psi \\ H \\ \Gamma, pc, l, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi_2\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^{\perp}\} \\ H_2\{l = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^{\perp} \\ \qquad\qquad |be| \\ \qquad\qquad \texttt{pushJP}\ l'; \\ \qquad\qquad \texttt{bnz}\ r_a,\ l_{else}[X]; \\ \qquad\qquad \texttt{jmp}\ l_{then}[X], \\ \\ \qquad l_{iJP} = \texttt{CODE}\langle\forall[X]\{sp : l'.\Sigma\} \mid pc\rangle^{\perp} \\ \qquad\qquad \texttt{jmpJP}\ l'[X]\} \\ \Sigma \end{bmatrix}$$

La traducción de la sentencia de iteración comienza en el bloque de código l_0 insertando en la pila el *junction point* l'. Luego el bloque de código l evalúa la expresión booleana y salta al bloque de código l_{bucle} si el resultado de la evaluación de la condición es *true*. Si evalúa en *false*, salta a la etiqueta l_{wJP} quien finalmente salta al *junction point* l'. Dentro del bloque de código l_{bucle}, que contiene la traducción de *stm*, al finalizar su ejecución vuelve al bloque de código l quien nuevamente realiza la evaluación de la condición.

$$\left|\frac{d}{\Delta, pc \vdash stm}\right| \begin{bmatrix} \Psi_1\{l = \texttt{CODE}\langle\forall[X]\{sp : l'.\Sigma\} \mid pc\rangle^{\perp}\} \\ H_1 \\ \Gamma, pc, l_{bucle}, l, l'.\Sigma \end{bmatrix} = \begin{bmatrix} \Psi_2 \\ H_2 \\ l'.\Sigma \end{bmatrix}$$

$$\left|\frac{\Delta \vdash be : pc \;\; \Delta, pc \vdash stm}{\Delta, pc \vdash while\ be\ do\ stm\ end}\right| \begin{bmatrix} \Psi \\ H \\ \Gamma, pc, l_0, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi_2\{l_0 = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^{\perp}, \\ \quad l_{iJP} = \texttt{CODE}\langle\forall[X]\{sp : l'.\Sigma\} \mid pc\rangle^{\perp}\} \\ H_2\{l_0 = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^{\perp} \\ \qquad\qquad \texttt{pushJP}\ l'; \\ \qquad\qquad \texttt{jmp}\ l[X], \\ \\ \quad l = \texttt{CODE}\langle\forall[X]\{sp : l'.\Sigma\} \mid pc\rangle^{\perp} \\ \qquad\qquad |be| \\ \qquad\qquad \texttt{bnz}\ r_a,\ l_{bucle}[X]; \\ \qquad\qquad \texttt{jmp}\ l_{wJP}[X], \\ \\ \quad l_{wJP} = \texttt{CODE}\langle\forall[X]\{sp : l'.\Sigma\} \mid pc\rangle^{\perp} \\ \qquad\qquad \texttt{jmpJP}\ l'[X]\} \\ \Sigma \end{bmatrix}$$

La traducción de las sentencias compuestas se muestra a continuación. Note que se traducirá primero stm_2, para tener el tipo de la etiqueta l_1 al momento de querer tipar la traducción de stm_1. Esto nos permite cumplir con lo definido inicialmente sobre l_{fin}.

$$\left|\frac{d_2}{\Gamma, pc \vdash stm_2}\right| \begin{bmatrix} \Psi \\ H \\ \Gamma, pc, l_1, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi_1 \\ H_1 \\ \Sigma \end{bmatrix} \quad \left|\frac{d_1}{\Delta, pc \vdash stm_1}\right| \begin{bmatrix} \Psi_1 \\ H_1 \\ \Gamma, pc, l, l_1, \Sigma \end{bmatrix} = \begin{bmatrix} \Psi_2 \\ H_2 \\ \Sigma \end{bmatrix}$$

$$\left|\frac{\Delta, pc \vdash stm_1 \quad \Delta, pc \vdash stm_2}{\Delta, pc \vdash stm_1; stm_2}\right| \begin{bmatrix} \Psi \\ H \\ \Gamma, pc, l, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi_2 \\ H_2 \\ \Sigma \end{bmatrix}$$

Finalmente, la regla de tipado $[\mathsf{T} - \mathsf{Sub}]$ no genera código, solo cambia el pc que se utilizará para tipar el resto de la derivación. H y Ψ no son modificadas.

$$\left|\frac{d}{\Delta, \mathsf{H} \vdash stm}\right| \begin{bmatrix} \Psi \\ H \\ \Gamma, \top, l, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi \\ H \\ \Sigma \end{bmatrix}$$

$$\left|\frac{\Delta, \mathsf{H} \vdash stm}{\Delta, \mathsf{L} \vdash stm}\right| \begin{bmatrix} \Psi \\ H \\ \Gamma, \perp, l, l', \Sigma \end{bmatrix} = \begin{bmatrix} \Psi \\ H \\ \Sigma \end{bmatrix}$$

3.2.3. Ejemplo

```
(pc = L) var a: L := 1 in
(pc = L)     var b: H := 0 in
(pc = L)         if b = 0
(pc = H)                 then b := 1
(pc = H)                 else b := 2
(pc = L)         end;
(pc = L)         a := 3
(pc = L)     end
(pc = L) end
```

Arbol de Derivación

$$\dfrac{\dfrac{}{\Delta \vdash 1 : \mathsf{L}}\,[\text{T-EConst}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta(b) = \mathsf{H}}{\Delta \vdash b : \mathsf{H}}\,[\text{T-EVar}]}{\Delta \vdash b = 0 : \mathsf{H}}\,[\text{T-BEIgual}] \quad \dfrac{\Delta(b) = \mathsf{H}}{\Delta, \mathsf{H} \vdash b := 1}\,[\text{T-AsigHigh}] \quad \dfrac{\Delta(b) = \mathsf{H}}{\Delta, \mathsf{H} \vdash b := 2}\,[\text{T-AsigHigh}]}{\Delta, \mathsf{H} \vdash \texttt{if } b = 0 \texttt{ then } b := 1 \texttt{ else } b := 2 \texttt{ end}}\,[\text{T-IF}]}{\Delta, \mathsf{L} \vdash \texttt{if } b = 0 \texttt{ then } b := 1 \texttt{ else } b := 2 \texttt{ end}}\,[\text{T-Sub}] \quad \dfrac{\dfrac{}{\Delta \vdash 3 : \mathsf{L}} \quad \Delta(a) = \mathsf{L}}{\Delta, \mathsf{L} \vdash a := 3}\,[\text{T-AsigLow}]}{\Delta[b \mapsto \mathsf{H}], \mathsf{L} \vdash \texttt{if } b = 0 \texttt{ then } b := 1 \texttt{ else } b := 2 \texttt{ end}; \; a := 3}\,[\text{T-Stm}]}{\Delta[a \mapsto \mathsf{L}], \mathsf{L} \vdash \texttt{var } b : \mathsf{H} := 0 \; in \; \texttt{if } b = 0 \texttt{ then } b := 1 \texttt{ else } b := 2 \texttt{ end}; \; a := 3 \texttt{ end}}\,[\text{T-DecHigh}]}{\Delta, \mathsf{L} \vdash \texttt{var } a : \mathsf{L} := 1 \; in \; \texttt{var } b : \mathsf{H} := 0 \; in \; \texttt{if } b = 0 \texttt{ then } b := 1 \texttt{ else } b := 2 \texttt{ end}; \; a := 3 \texttt{ end end}}\,[\text{T-DecLow}]}{\Delta \vdash \texttt{var } a : \mathsf{L} := 1 \; in \; \texttt{var } b : \mathsf{H} := 0 \; in \; \texttt{if } b = 0 \texttt{ then } b := 1 \texttt{ else } b := 2 \texttt{ end}; \; a := 3 \texttt{ end end}}\,[\text{T-Prog}]$$

Traducción en SecTAL

$Lmain$: $\texttt{CODE}\langle\forall[X]\{sp : Halt.X\}|\bot\rangle^{\bot}$ % Generado por [T-Prog]
$\texttt{jmp}\ l[X]$

l: $\texttt{CODE}\langle\forall[X]\{sp : Halt.X\}|\bot\rangle^{\bot}$ % Generado por [T-DecLow]
$\texttt{mov}\ r_a, 1$ % expresion 1 en r_a
$\texttt{salloc}\ 1$ % aloco espacio para a
$\texttt{sst}\ sp[0], r_a$ % a := 1
$\texttt{jmp}\ l1[X]$

$l1$: $\texttt{CODE}\langle\forall[X]\{sp : int^{\bot}.Halt.X\}|\bot\rangle^{\bot}$ % Generado por [T-DecHigh]
$\texttt{mov}\ r_a, 0$ % expresion 0 en r_a
$\texttt{salloc}\ 1$ % aloco espacio para b
$\texttt{sstNsl}\ \tau(\mathsf{H}), sp[0], r_a$ % b := 0
$\texttt{jmp}\ l2[X]$

$l2$: $\texttt{CODE}\langle\forall[X]\{sp : int^{\top}.int^{\bot}.Halt.X\}|\bot\rangle^{\bot}$ % Generado por [T-IF]
$\texttt{sld}\ r_a, sp[0]$ % expresion b en r_a
$\texttt{pushJP}\ lprima$
$\texttt{bnz}\ r_a, l_{else}[X]$ % el salto se hace sobre r_a que es H
$\texttt{jmp}\ l_{then}[X]$

l_{then}: $\texttt{CODE}\langle\forall[X]\{sp : lprima.int^{\top}.int^{\bot}.Halt.X\}|\top\rangle^{\bot}$ % Generado por [T-AsigHigh]
$\texttt{mov}\ r_a, 1$ % expresion 1 en r_a
$\texttt{sst}\ sp[1], r_a$ % b := 1
$\texttt{jmp}\ l_{iJP}[X]$

l_{else}: $\texttt{CODE}\langle\forall[X]\{sp : lprima.int^{\top}.int^{\bot}.Halt.X\}|\top\rangle^{\bot}$ % Generado por [T-AsigHigh]
$\texttt{mov}\ r_a, 2$ % expresion 2 en r_a
$\texttt{sst}\ sp[1], r_a$ % b := 2
$\texttt{jmp}\ l_{iJP}[X]$

l_{iJP}: $\texttt{CODE}\langle\forall[X]\{sp : lprima.int^{\top}.int^{\bot}.Halt.X\}|\top\rangle^{\bot}$ % Generado por [T-IF]
$\texttt{jmpJP}\ lprima[X]$

$lprima$: $\texttt{CODE}\langle\forall[X]\{sp : int^{\top}.int^{\bot}.Halt.X\}|\bot\rangle^{\bot}$ % Generado por [T-AsigLow]
$\texttt{mov}\ r_a, 3$ % expresion 3 en r_a
$\texttt{sst}\ sp[1], r_a$ % a := 3
$\texttt{jmp}\ l7[X]$

$l7$: `CODE`$\langle\forall[X]\{sp : int^{\top}.int^{\bot}.Halt.X\}|\bot\rangle^{\bot}$ % Generado por [T-DecHigh]
`sfree` 1
`jmp` $l8[X]$

$l8$: `CODE`$\langle\forall[X]\{sp : int^{\bot}.Halt.X\}|\bot\rangle^{\bot}$ % Generado por [T-DecLow]
`sfree` 1
`jmp` $l_{halt}[X]$

l_{halt}: `CODE`$\langle\forall[X]\{sp : Halt.X\}|\bot\rangle^{\bot}$ % Generado por [T-Prog]
`halt`

3.2.4. Preservación de Tipos

La función de compilación presentada será correcta si la traducción de un programa WHILE a un programa SECTAL preserva tipos. Comenzaremos definiendo y demostrando los lemas para la traducción de expresiones y sentencias para luego finalizar con la traducción de un programa.

Lema 3 (traducción de expresiones). a : Si $\Delta \sim \Gamma(sp)$, $\Delta \vdash e : t$ y
$\Theta \mid \Gamma\{r_a : \tau(t) \sqcup pc, sp : \Sigma\} \mid pc \rhd_\Psi B$, entonces, $\Theta \mid \Gamma\{r_a :?, sp : \Sigma\} \mid pc \rhd_\Psi |e|; B$.

b : Si $\Delta \sim \Gamma(sp)$, $\Delta \vdash be : t$ y $\Theta \mid \Gamma\{r_a : \tau(t) \sqcup pc, sp : \Sigma\} \mid pc \rhd_\Psi B$, entonces, $\Theta \mid \Gamma\{r_a :?, sp : \Sigma\} \mid pc \rhd_\Psi |be|; B$.

Donde r_a :? significa que el registro r_a podría existir antes de la traducción de $|e|$ con cualquier otro tipo.

Demostración. La demostración es por inducción estructural en e y be:

Caso $e = i$**.** $|i| = mov\ r_a,\ i$, las constantes son de tipo L y $\tau(\mathsf{L}) = \tau(t) = \bot$. Entonces tenemos que demostrar que $\Theta \mid \Gamma\{sp : \Sigma\} \mid pc \rhd_\Psi \texttt{mov}\ r_a,\ i; B$ sabiendo que $\Theta \mid \Gamma\{r_a : \bot \sqcup pc, sp : \Sigma\} \mid pc \rhd_\Psi B$. Aplicando [T-Mov] obtenemos:

$$\frac{\Theta \mid \Gamma\{sp : \Sigma\} \rhd_\Psi i : \bot\ opnd \qquad \Theta \mid \Gamma\{r_a : \bot \sqcup pc, sp : \Sigma\} \mid pc \rhd_\Psi B}{\Theta \mid \Gamma\{sp : \Sigma\} \mid pc \rhd_\Psi \texttt{mov}\ r_a,\ i; B}\ \text{[T-Mov]}$$

Dado que $\Theta \mid \Gamma\{sp : \Sigma\} \rhd_\Psi i : \bot\ opnd$ es un operando bien tipado y sabiendo que $\Theta \mid \Gamma\{r_a : \bot \sqcup pc, sp : \Sigma\} \mid pc \rhd_\Psi B$, concluimos.

Caso $e = x$**.** $|x| = \texttt{sld}\ r_a,\ sp[offset(x, \Sigma)]$, sabiendo que $\Delta \sim \Gamma(sp)$, tenemos $\tau(t) = \tau(\Delta(x)) = \Gamma(sp)[offset(x, \Sigma)]$. Aplicando [T-Sld] obtenemos:

$$\frac{\Gamma(sp) = \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(t) \cdot \Sigma' \qquad \Theta \mid \Gamma\{r_a : \tau(\Delta(x)) \sqcup pc, sp : \Sigma\} \mid pc \rhd_\Psi B}{\Theta \mid \Gamma\{sp : \Sigma\} \mid pc \rhd_\Psi \texttt{sld}\ r_a,\ sp[i]; B}\ \text{[T-Sld]}$$

Sabiendo que $\Theta \mid \Gamma\{r_a : \tau(t) \sqcup pc, sp : \Sigma\} \mid pc \ \rhd_\Psi B$, concluimos.

Caso $e = e_1 + e_2$**.** $|e_1 + e_2| \ = \ |e_1|;\ \texttt{salloc}\ 1;\ \texttt{sst}\ sp[0],\ r_a;\ |e_2|;\ \texttt{sld}\ r_b,\ sp[0];\ \texttt{add}\ r_a,\ r_b,\ r_a;\ \texttt{sfree}\ 1.$

Vamos a dividir la prueba en dos partes. Primero vamos a probar que:
$\Theta \mid \Gamma\{r_a : \tau(t_2) \sqcup pc, sp : (\tau(t_1) \sqcup pc).\Sigma'\} \mid pc \ \rhd_\Psi \texttt{sld}\ r_b,\ sp[0];\ \texttt{add}\ r_a,\ r_b,\ r_a;\ \texttt{sfree}\ 1; B'$, asumiendo que: $\Theta \mid \Gamma\{r_a : \tau(t_2) \sqcup \tau(t_1) \sqcup pc, r_b : \tau(t_1) \sqcup pc, sp : \Sigma'\} \mid pc \ \rhd_\Psi B'$. Donde, $\Delta \vdash e_1 : t_1$ y $\Delta \vdash e_2 : t_2$.

$$\text{d2} = \frac{\Gamma(sp) = (\tau(t_1) \sqcup pc).\Sigma' \qquad \Theta \mid \Gamma[sp := \Sigma'] \mid pc \ \rhd_\Psi B'}{\Theta \mid \Gamma\{r_a : \tau(t_2) \sqcup \tau(t_1) \sqcup pc, r_b : \tau(t_1) \sqcup pc, sp : (\tau(t_1) \sqcup pc).\Sigma'\} \mid pc \ \rhd_\Psi \texttt{sfree}\ 1; B'}\ \text{[T-Sfree]}$$

$$\text{d1} = \frac{\dfrac{}{\Theta \mid \Gamma \rhd_\Psi r_a : \tau(t_2) \sqcup pc\ opnd}\ \text{[T-RegOp]} \qquad \dfrac{}{\Theta \mid \Gamma \rhd_\Psi r_b : \tau(t_1) \sqcup pc\ opnd}\ \text{[T-RegOp]} \qquad d2}{\Theta \mid \Gamma\{r_a : \tau(t_2) \sqcup pc, r_b : \tau(t_1) \sqcup pc, sp : (\tau(t_1) \sqcup pc).\Sigma'\} \mid pc \ \rhd_\Psi \texttt{add}\ r_a,\ r_b,\ r_a;\ \texttt{sfree}\ 1; B'}\ \text{[T-Arith]}$$

$$\frac{\Gamma(sp) = (\tau(t_1) \sqcup pc).\Sigma' \qquad d1}{\Theta \mid \Gamma\{r_a : \tau(t_2) \sqcup pc, sp : (\tau(t_1) \sqcup pc).\Sigma'\} \mid pc \ \rhd_\Psi \texttt{sld}\ r_b,\ sp[0];\ \texttt{add}\ r_a,\ r_b,\ r_a;\ \texttt{sfree}\ 1; B'}\ \text{[T-Sld]}$$

Sabiendo que $\Theta \mid \Gamma\{r_a : \tau(t_2) \sqcup \tau(t_1) \sqcup pc, r_b : \tau(t_1) \sqcup pc, sp : \Sigma'\} \mid pc \ \rhd_\Psi B'$, concluimos. Luego por H.I. con:

$|e| = |e_2|$

$t = t_2$

$\Sigma = (\tau(t_1) \sqcup pc).\Sigma'$

$B = \texttt{sld}\ r_b,\ sp[0];\ \texttt{add}\ r_a,\ r_b,\ r_a;\ \texttt{sfree}\ 1; B'$

entonces, $\Theta \mid \Gamma\{r_a : \tau(t_1) \sqcup pc, sp : (\tau(t_1) \sqcup pc).\Sigma'\} \mid pc \ \rhd_\Psi |e_2|;\ \texttt{sld}\ r_b,\ sp[0];\ \texttt{add}\ r_a,\ r_b,\ r_a;\ \texttt{sfree}\ 1; B'$.

Finalmente vamos a probar que: $\Theta \mid \Gamma\{r_a : \tau(t_1) \sqcup pc, sp : \Sigma'\} \mid pc \ \rhd_\Psi \ \texttt{salloc}\ 1;\ \texttt{sst}\ sp[0],\ r_a; B''$,
donde $B'' \ = \ |e_2|;\ \texttt{sld}\ r_b,\ sp[0];\ \texttt{add}\ r_a,\ r_b,\ r_a;\ \texttt{sfree}\ 1; B'$

$$\text{d1} = \dfrac{\Gamma(sp) = ns.\Sigma' \qquad \dfrac{}{\Theta \mid \Gamma \rhd_\Psi \ r_a : \tau(t_1) \sqcup pc \ opnd}\ [\text{T-RegOp}] \qquad \Theta \mid \Gamma[sp := (\tau(t_1) \sqcup pc).\Sigma'] \ \mid pc \ \rhd_\Psi \ B''}{\Theta \mid \Gamma\{r_a : \tau(t_1) \sqcup pc, sp : ns.\Sigma'\} \ \mid pc \ \rhd_\Psi \texttt{sst}\ sp[0],\ r_a; B''}\ [\text{T-SstNs}]$$

$$\dfrac{\Gamma(sp) = \Sigma' \qquad d1}{\Theta \mid \Gamma\{r_a : \tau(t_1) \sqcup pc, sp : \Sigma'\} \ \mid pc \ \rhd_\Psi \texttt{salloc}\ 1;\ \texttt{sst}\ sp[0],\ r_a; B''}\ [\text{T-Salloc}]$$

Sabiendo que $\Theta \mid \Gamma\{r_a : \tau(t_1) \sqcup pc, sp : (\tau(t_1) \sqcup pc).\Sigma'\} \ \mid pc \ \rhd_\Psi B''$, concluimos. Luego por H.I. con:

$|e| = |e_1|$

$t = t_1$

$\Sigma = \Sigma'$

$B = \texttt{salloc}\ 1;\ \texttt{sst}\ sp[0],\ r_a; B''$

entonces, $\Theta \mid \Gamma\{sp : \Sigma'\} \ \mid pc \ \rhd_\Psi |e_1|;\ \texttt{salloc}\ 1;\ \texttt{sst}\ sp[0],\ r_a; B''$

Caso $be = true$**.** $|true| = \texttt{mov}\ r_a,\ 1$, las constantes son de tipo L y $\tau(\mathsf{L}) = \tau(t) = \bot$. Entonces tenemos que demostrar $\Theta \mid \Gamma\{sp : \Sigma\} \mid pc \rhd_\Psi \ \texttt{mov}\ r_a,\ 1; B$, sabiendo que $\Theta \mid \Gamma\{r_a : \bot \sqcup pc, sp : \Sigma\} \mid pc \ \rhd_\Psi B$. Aplicando [T-Mov] obtenemos:

$$\dfrac{\Theta \mid \Gamma\{sp : \Sigma\} \ \rhd_\Psi 1 : \bot \ opnd \qquad \Theta \mid \Gamma\{r_a : \bot \sqcup pc, sp : \Sigma\} \ \mid pc \rhd_\Psi B}{\Theta \mid \Gamma\{sp : \Sigma\} \ \mid pc \ \rhd_\Psi \ \texttt{mov}\ r_a,\ 1; B}\ [\text{T-Mov}]$$

Dado que $\Theta \mid \Gamma\{sp : \Sigma\} \rhd_\Psi 1 : \bot \ opnd$ es un operando bien tipado y sabiendo que $\Theta \mid \Gamma\{r_a : \bot \sqcup pc, sp : \Sigma\} \mid pc \rhd_\Psi B$, concluimos.

Caso $be = false$**.** Idem caso anterior.

Caso $be = |e_1 = 0|$**.** $|e_1 = 0| \ = \ |e_1|$. Por Lema 3 (a), tenemos que $\Theta \mid \Gamma\{sp : \Sigma\} \ \mid pc \ \rhd_\Psi |e_1|; B$

□

Lema 4 (traducción de sentencias). Si $\Delta \sim \Gamma(sp)$, $\Delta, pc_s \vdash stm$,

$$\left|\frac{d}{\Delta, pc_s \vdash stm}\right| \quad \begin{bmatrix} \Psi \\ H \\ \Gamma, pc, l_{inicio}, l_{fin}, \Sigma \end{bmatrix} = \begin{bmatrix} \Psi' \\ H' \\ \Sigma \end{bmatrix}, \ \Psi(l_{fin}) = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^{\perp},$$

$\triangleright H : \Psi$, $pc_s = pc$, entonces, $\triangleright H' : \Psi'$.

Demostración. Prueba por inducción sobre la derivación de la traducción de *stm*. Cada una de las reglas de traducción agrega uno o más bloques de código a H, generando H'. Vamos a probar que $\triangleright H' : \Psi'$, sabiendo que $\triangleright H : \Psi$. Para esto tenemos que probar que cada uno de los bloques agregados a H tipan.

Caso T-AsigLow. La traducción de la asignación agrega el bloque de código l a H.

$$\text{d2} = \frac{\dfrac{1 \geq 0}{\triangleright_\Psi \{r_a : \tau(\mathsf{L}), sp : \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{L}) \cdot \Sigma'\} \leq \{sp : \Sigma\}}\ \text{[ST-RegBank]} \qquad \tau(\mathsf{L}) \sqsubseteq \tau(\mathsf{L})}{\Theta \triangleright \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid \tau(\mathsf{L})\rangle^{\perp} \leq \Theta \triangleright \texttt{CODE}\langle\forall[X]\{r_a : \tau(\mathsf{L}), sp : \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{L}) \cdot \Sigma'\} \mid \tau(\mathsf{L})\rangle^{\perp}}\ \text{[ST-Code]}$$

$$\text{d1} = \frac{\dfrac{}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{L}) \cdot \Sigma'\} \ \triangleright_\Psi l_{fin} : \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid \tau(\mathsf{L})\rangle^{\perp}\ opnd}\ \text{[T-RegOp]} \qquad d2}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{L}) \cdot \Sigma'\} \mid \tau(\mathsf{L}) \ \triangleright_\Psi \texttt{jmp}\ l_{fin}[X]\ blk}$$

$$\frac{d1 \qquad \tau(\mathsf{L}) \sqcup \tau(\mathsf{L}) \sqsubseteq \tau(\mathsf{L}) \qquad \Gamma(sp) = \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{L}) \cdot \Sigma' \qquad \Theta \mid \Gamma \triangleright_\Psi r_a : \tau(\mathsf{L})\ opnd}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : \Sigma\} \mid \tau(\mathsf{L}) \ \triangleright_\Psi \texttt{sst}\ sp[offset(x, \Sigma)],\ r_a; \texttt{jmp}\ l_{fin}[X]\ blk}\ \text{[T-Sst]}$$

Luego por Lema 3 (a) podemos concluir que: $\Theta \mid \Gamma\{sp : \Sigma\} \mid \tau(\mathsf{L}) \ \triangleright_\Psi \ |e|; \texttt{sst}\ sp[offset(x, \Sigma)],\ r_a; \texttt{jmp}\ l_{fin}[X]\ blk$

Caso T-AsigHigh. Procedemos similar al caso anterior.

$$\text{d2} = \dfrac{\dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(t), sp : \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{H}) \cdot \Sigma'\} \leq \{sp : \Sigma\}}\ \text{[ST-RegBank]} \qquad pc \sqsubseteq\ pc}{\Theta \rhd \texttt{CODE}\langle \forall [X] \{sp : \Sigma\} \mid pc \rangle^\perp \leq \Theta \rhd \texttt{CODE}\langle \forall [X] \{r_a : \tau(t), sp : \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{H}) \cdot \Sigma'\} \mid pc \rangle^\perp}\ \text{[ST-Code]}$$

$$\text{d1} = \dfrac{\dfrac{}{\Theta \mid \Gamma \{r_a : \tau(\mathsf{L}), sp : \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{H}) \cdot \Sigma'\} \rhd_\Psi l_{fin} : \texttt{CODE}\langle \forall [X] \{sp : \Sigma\} \mid pc \rangle^\perp\ opnd}\ \text{[T-RegOp]} \qquad d2}{\Theta \mid \Gamma \{r_a : \tau(\mathsf{L}), sp : \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{H}) \cdot \Sigma'\} \mid pc \rhd_\Psi \texttt{jmp}\ l_{fin}[X]\ blk}$$

$$\dfrac{d1 \qquad \Gamma(sp) = \sigma_0 \cdot ... \cdot \sigma_{i-1} \cdot \tau(\mathsf{H}) \cdot \Sigma' \qquad pc \sqcup \tau(t) \sqsubseteq \tau(\mathsf{H}) \qquad \Theta \mid \Gamma \rhd_\Psi r_a : \tau(t)\ opnd}{\Theta \mid \Gamma \{r_a : \tau(t), sp : \Sigma\} \mid pc \rhd_\Psi \texttt{sst}\ sp[\mathit{offset}(x, \Sigma)],\ r_a; \texttt{jmp}\ l_{fin}[X]\ blk}\ \text{[T-Sst]}$$

Luego por Lema 3 (a) podemos concluir que: $\Theta \mid \Gamma \{sp : \Sigma\} \mid pc \rhd_\Psi |e|; \texttt{sst}\ sp[\mathit{offset}(x, \Sigma)],\ r_a; \texttt{jmp}\ l_{fin}[X]\ blk$

Caso T-DecLow. La traducción de la declaración agrega dos bloques de códigos. Probaremos ambos por separado, empezando por el bloque l y donde $\Psi(l_{fin}) = \Psi(l_1) = \texttt{CODE}\langle \forall [X] \{sp : \tau(\mathsf{L}) \cdot \Sigma\} \mid \tau(\mathsf{L}) \rangle^\perp$

$$\text{d2} = \dfrac{\dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(\mathsf{L}), sp : \tau(\mathsf{L}) \cdot \Sigma\} \leq \{sp : \tau(\mathsf{L}) \cdot \Sigma\}}\ \text{[ST-RegBank]} \qquad \tau(\mathsf{L}) \sqsubseteq \tau(\mathsf{L})}{\Theta \rhd \texttt{CODE}\langle \forall [X] \{sp : \tau(\mathsf{L}) \cdot \Sigma\} \mid \tau(\mathsf{L}) \rangle^\perp \leq \texttt{CODE}\langle \forall [X] \{r_a : \tau(\mathsf{L}), sp : \tau(\mathsf{L}) \cdot \Sigma\} \mid \tau(\mathsf{L}) \rangle^\perp}\ \text{[ST-Code]}$$

$$\text{d1} = \dfrac{\Theta \mid \Gamma \{r_a : \tau(\mathsf{L}), sp : \tau(\mathsf{L}) \cdot \Sigma\} \rhd_\Psi l_1 : \texttt{CODE}\langle \forall [X] \{sp : \tau(\mathsf{L}) \cdot \Sigma\} \mid \tau(\mathsf{L}) \rangle^\perp\ opnd \qquad d2}{\Theta \mid \Gamma \{r_a : \tau(\mathsf{L}), sp : \tau(\mathsf{L}) \cdot \Sigma\} \mid \tau(\mathsf{L}) \rhd_\Psi \texttt{jmp}\ l_1[X]\ blk}\ \text{[T-RegOp]}$$

$$\dfrac{\dfrac{d1 \qquad \Gamma(sp) = ns \cdot \Sigma \qquad \Theta \mid \Gamma \rhd_\Psi r_a : \tau(\mathsf{L})\ opnd}{\Theta \mid \Gamma \{r_a : \tau(\mathsf{L}), sp : ns \cdot \Sigma\} \mid \tau(\mathsf{L}) \rhd_\Psi \texttt{sst}\ sp[0],\ r_a; \texttt{jmp}\ l_1[X]\ blk}\ \text{[T-SstNs]}}{\Theta \mid \Gamma \{r_a : \tau(\mathsf{L}), sp : \Sigma\} \mid \tau(\mathsf{L}) \rhd_\Psi \texttt{salloc}\ 1;\ \texttt{sst}\ sp[0],\ r_a; \texttt{jmp}\ l_1[X]\ blk}\ \text{[T-Salloc]}$$

Luego por Lema 3 (a) podemos concluir que: $\Theta \mid \Gamma \{sp : \Sigma\} \mid \tau(\mathsf{L}) \rhd_\Psi |e|; \texttt{salloc}\ 1; \texttt{sst}\ sp[0],\ r_a; \texttt{jmp}\ l_1[X]\ blk$

Ahora vamos a seguir con el bloque l_2.

$$\text{d2} = \dfrac{\dfrac{0 \geq 0}{\rhd_\Psi \{sp : \Sigma\} \leq \{sp : \Sigma\}}\,[\text{ST-RegBank}] \qquad \tau(\mathsf{L}) \sqsubseteq\ \tau(\mathsf{L})}{\Theta \rhd \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid \tau(\mathsf{L})\rangle^\perp \leq \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid \tau(\mathsf{L})\rangle^\perp}\,[\text{ST-Code}]$$

$$\text{d1} = \dfrac{\dfrac{}{\Theta \mid \Gamma\{sp : \Sigma\}\ \rhd_\Psi l_{fin} : \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid \tau(\mathsf{L})\rangle^\perp\ opnd \qquad d2}\,[\text{T-RegOp}]}{\Theta \mid \Gamma\{sp : \Sigma\} \mid \tau(\mathsf{L})\ \rhd_\Psi\ \texttt{jmp}\ l_{fin}[X]\ blk}$$

$$\dfrac{d1 \qquad \Gamma(sp) = \tau(\mathsf{L}) \cdot \Sigma}{\Theta \mid \Gamma\{sp : \tau(\mathsf{L}) \cdot \Sigma\}\ \mid \tau(\mathsf{L})\ \rhd_\Psi \texttt{sfree}\ 1;\ \texttt{jmp}\ l_{fin}[X]\ blk}\,[\text{T-Free}]$$

Caso T-DecHigh. Similar al caso anterior, comenzamos por el bloque l, y donde $\Psi(l_{fin}) = \Psi(l_1) = \texttt{CODE}\langle\forall[X]\{sp : \tau(\mathsf{H}) \cdot \Sigma\} \mid pc\rangle^\perp$.

$$\text{d3} = \dfrac{\dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(t), sp : \tau(\mathsf{H}) \cdot \Sigma\} \leq \{sp : \tau(\mathsf{H}) \cdot \Sigma\}}\,[\text{ST-RegBank}] \qquad pc \sqsubseteq\ pc}{\Theta \rhd \texttt{CODE}\langle\forall[X]\{sp : \tau(\mathsf{H}) \cdot \Sigma\} \mid pc\rangle^\perp \leq \Theta \rhd \texttt{CODE}\langle\forall[X]\{r_a : \tau(t), sp : \tau(\mathsf{H}) \cdot \Sigma\} \mid pc\rangle^\perp}\,[\text{ST-Code}]$$

$$\text{d2} = \dfrac{\dfrac{}{\Theta \mid \Gamma\{r_a : \tau(t), sp : \tau(\mathsf{H}) \cdot \Sigma\}\ \rhd_\Psi l_1 : \texttt{CODE}\langle\forall[X]\{sp : \tau(\mathsf{H}) \cdot \Sigma\} \mid pc\rangle^\perp opnd \qquad d3}\,[\text{T-RegOp}]}{\Theta \mid \Gamma\{r_a : \tau(t), sp : \tau(\mathsf{H}) \cdot \Sigma\} \mid pc\ \rhd_\Psi\ \texttt{jmp}\ l_1[X]\ blk}$$

$$\text{d1} = \dfrac{d2 \qquad \Theta \mid \Gamma \rhd_\Psi r_a : \tau(t)\ opnd \qquad \Gamma(sp) = ns \cdot \Sigma}{\Theta \mid \Gamma\{r_a : \tau(t), sp : ns \cdot \Sigma\}\ \mid pc\ \rhd_\Psi \texttt{sstNsl}\ \tau(\mathsf{H}),\ sp[0],\ r_a; \texttt{jmp}\ l_1[X]\ blk}\,[\text{T-SstNsl}]$$

$$\dfrac{d1 \qquad \Gamma(sp) = \Sigma}{\Theta \mid \Gamma\{r_a : \tau(t), sp : \Sigma\}\ \mid pc\ \rhd_\Psi \texttt{salloc}\ 1;\ \texttt{sstNsl}\ \tau(\mathsf{H}),\ sp[0],\ r_a; \texttt{jmp}\ l_1[X]\ blk}\,[\text{T-Salloc}]$$

Luego por Lema 3 (a) $\Theta \mid \Gamma\{sp : \Sigma\}\ \mid pc\ \rhd_\Psi |e|;\ \texttt{salloc}\ 1;\ \texttt{sstNsl}\ \tau(\mathsf{H}),\ sp[0],\ r_a; \texttt{jmp}\ l_1[X]\ blk$.

Continuamos con el bloque l_2.

$$\text{d3} = \dfrac{\dfrac{0 \geq 0}{\rhd_\Psi \{sp : \Sigma\} \leq \{sp : \Sigma\}}\,[\text{ST-RegBank}] \qquad pc \sqsubseteq\ pc}{\Theta \rhd \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\perp \leq \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\perp}\,[\text{ST-Code}]$$

$$\text{d2} = \dfrac{\dfrac{}{\Theta \mid \Gamma\{sp : \Sigma\}\ \rhd_\Psi l_{fin} : \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\perp\ opnd}\,[\text{T-RegOp}] \qquad d3}{\Theta \mid \Gamma\{sp : \Sigma\} \mid pc\ \rhd_\Psi\ \texttt{jmp}\ l_{fin}[X]\ blk}$$

$$\dfrac{d1 \qquad \Gamma(sp) = \tau(\mathsf{H}) \cdot \Sigma}{\Theta \mid \Gamma\{sp : \tau(\mathsf{H}) \cdot \Sigma\}\ \mid pc\ \rhd_\Psi \texttt{sfree}\ 1;\ \texttt{jmp}\ l_{fin}[X]\ blk}\,[\text{T-Free}]$$

Caso T-IF para $\Delta(be) = \mathsf{L}$. La traducción de una sentencia de selección genera dos bloques de código. Comenzaremos con el bloque l con $\Psi(l_{fin}) = \Psi(l_{else}) = \Psi(l_{then}) = \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp$.

$$\dfrac{\dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\,[\text{ST-RegBank}] \qquad pc \sqcup \tau(\mathsf{L}) \sqsubseteq\ pc}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\} \mid pc \sqcup \tau(\mathsf{L})\rangle^\perp}\,[\text{ST-Code}]$$

$$\text{d2} = \dfrac{\dfrac{}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\}\ \rhd_\Psi l_{then} : \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp\ opnd}\,[\text{T-RegOp}]}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\}\ \mid pc \sqcup \tau(\mathsf{L})\ \rhd_\Psi\ \texttt{jmp}\ l_{then}[X]\ blk}\,[\text{T-Jmp}]$$

$$\dfrac{pc \sqcup \tau(\mathsf{L}) \sqcup \perp \sqsubseteq\ pc \qquad \dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\,[\text{ST-RegBank}]}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\} \mid pc \sqcup \tau(\mathsf{L}) \sqcup \perp\rangle^\perp}\,[\text{ST-Code}]$$

$$\text{d1} = \dfrac{\dfrac{}{\Theta \mid \Gamma \rhd_\Psi l_{else} : \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp}\,[\text{T-RegOp}] \qquad \dfrac{}{\Theta \mid \Gamma \rhd_\Psi r_a : \tau(\mathsf{L})\ opnd}\,[\text{T-RegOp}] \qquad d2}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\}\ \mid pc\ \rhd_\Psi \texttt{bnz}\ r_a,\ l_{else}[X]; \texttt{jmp}\ l_{then}[X]\ blk}\,[\text{T-CondBranch}]$$

$$\dfrac{d1 \qquad \Psi(l') = \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\perp \qquad pc \sqsubseteq pc \qquad \Gamma(sp) = \Sigma}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : \Sigma\}\ \mid pc\ \rhd_\Psi \texttt{pushJP}\ l';\ \texttt{bnz}\ r_a,\ l_{else}[X]; \texttt{jmp}\ l_{then}[X]\ blk}\,[\text{T-Push}]$$

Luego por Lema 3 (b), $\Theta \mid \Gamma\{sp : \Sigma\} \mid pc \rhd_\Psi |be|;\ \texttt{pushJP}\ l';\ \texttt{bnz}\ r_a,\ l_{else}[X]; \texttt{jmp}\ l_{then}[X]\ blk$.

Continuamos con el bloque de código l_{iJP}.

$$\frac{\bot \sqsubseteq\ pc \qquad \dfrac{0 \geq 0}{\rhd_\Psi \{sp : \Sigma\} \leq \{sp : \Sigma\}}\ \text{[T-RegBank]}}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{\Gamma[sp := \Sigma]\} \mid \bot\rangle^\bot}$$

$$\frac{\Theta \rhd X\ \texttt{ok} \qquad \Gamma'(sp) = \Sigma \qquad \Gamma(sp) = l' \cdot \Sigma \qquad \Psi(l') = \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\bot}{\Theta \mid \Gamma\{sp : l' \cdot \Sigma\} \mid pc \rhd_\Psi \texttt{jmpJP}\ l'[X]\ blk}\ \text{[T-JmpJP]}$$

Caso T-IF para $\Delta(be) = \mathsf{H}$. Similar al caso anterior, comenzaremos con el bloque l con $\Psi(l_{fin}) = \Psi(l_{else}) = \Psi(l_{then}) = \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid \top\rangle^\bot$.

$$\frac{\dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\ \text{[ST-RegBank]} \qquad pc \sqcup \tau(\mathsf{H}) \sqsubseteq\ \top}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid \top\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \mid pc \sqcup \tau(\mathsf{H})\rangle^\bot}\ \text{[ST-Code]}$$

$$\text{d2} = \frac{\dfrac{}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \rhd_\Psi l_{then} : \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid \top\rangle^\bot\ opnd}\ \text{[T-RegOp]}}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \mid pc \sqcup \tau(\mathsf{H}) \rhd_\Psi \texttt{jmp}\ l_{then}[X]\ blk}\ \text{[T-Jmp]}$$

$$\frac{pc \sqcup \tau(\mathsf{H}) \sqcup \bot \sqsubseteq\ \top \qquad \dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\ \text{[ST-RegBank]}}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid \top\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \mid pc \sqcup \tau(\mathsf{H}) \sqcup \bot\rangle^\bot}\ \text{[ST-Code]}$$

$$\text{d1} = \frac{\dfrac{}{\Theta \mid \Gamma \rhd_\Psi l_{else} : \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid \top\rangle^\bot}\ \text{[T-RegOp]} \qquad \dfrac{}{\Theta \mid \Gamma \rhd_\Psi r_a : \tau(\mathsf{H})\ opnd}\ \text{[T-RegOp]} \qquad d2}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \mid pc \rhd_\Psi \texttt{bnz}\ r_a,\ l_{else}[X]; \texttt{jmp}\ l_{then}[X]\ blk}\ \text{[T-CondBranch]}$$

$$\frac{d1 \qquad \Psi(l') = \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\bot \qquad pc \sqsubseteq pc \qquad \Gamma(sp) = \Sigma}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{H}), sp : \Sigma\} \mid pc \rhd_\Psi \texttt{pushJP}\ l';\ \texttt{bnz}\ r_a,\ l_{else}[X]; \texttt{jmp}\ l_{then}[X]\ blk}\ \text{[T-Push]}$$

Luego por Lema 3 (b), $\Theta \mid \Gamma\{sp : \Sigma\} \mid pc \rhd_\Psi |be|;\ \texttt{pushJP}\ l';\ \texttt{bnz}\ r_a,\ l_{else}[X]; \texttt{jmp}\ l_{then}[X]\ blk$.

Continuamos con el bloque de código l_{iJP}.

$$\frac{\bot \sqsubseteq\ pc \qquad \dfrac{0 \geq 0}{\rhd_\Psi\{sp : \Sigma\} \leq \{sp : \Sigma\}}\ \text{[T-RegBank]}}{\rhd_\Psi \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle\forall[X]\{\Gamma[sp := \Sigma]\} \mid \bot\rangle^\bot}$$

$$\frac{\Theta \rhd X\ \mathsf{ok} \qquad \Gamma'(sp) = \Sigma \qquad \Gamma(sp) = l' \cdot \Sigma \qquad \Psi(l') = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^\bot}{\Theta \mid \Gamma\{sp : l' \cdot \Sigma\} \mid pc \rhd_\Psi \texttt{jmpJP}\ l'[X]\ blk}\ \text{[T-JmpJP]}$$

Caso T-WHILE para $\Delta(be) = \mathsf{L}$. La traducción de una sentencia de iteración genera dos nuevos bloques de código. Comenzaremos con el bloque l_0 con $\Psi(l_{fin}) = \Psi(l) = \texttt{CODE}\langle\forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\bot$.

$$\frac{\dfrac{0 \geq 0}{\rhd_\Psi\{sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\ \text{[ST-RegBank]} \qquad pc \sqsubseteq\ pc}{\rhd_\Psi \texttt{CODE}\langle\forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle\forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\bot}\ \text{[ST-Code]}$$

$$\text{d1} = \frac{\dfrac{}{\Theta \mid \Gamma\{sp : l' \cdot \Sigma\} \rhd_\Psi l : \texttt{CODE}\langle\forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\bot opnd}\ \text{[T-RegOp]}}{\Theta \mid \Gamma\{sp : l' \cdot \Sigma\} \mid pc \rhd_\Psi \texttt{jmp}\ l[X]\ blk}\ \text{[T-Jmp]}$$

$$\frac{d1 \qquad \Psi(l') = \texttt{CODE}\langle\forall[X]\{sp : \Sigma\} \mid pc\rangle^\bot \qquad pc \sqsubseteq pc \qquad \Gamma(sp) = \Sigma}{\Theta \mid \Gamma\{sp : \Sigma\} \mid pc \rhd_\Psi \texttt{pushJP}\ l'[X]; \texttt{jmp}\ l[X]\ blk}\ \text{[T-Push]}$$

Seguimos con el bloque l.

$$\dfrac{\dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\,[\text{ST-RegBank}] \qquad pc \sqcup \tau(\mathsf{L}) \sqsubseteq\ pc}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\} \mid pc \sqcup \tau(\mathsf{L})\rangle^\perp}\,[\text{ST-Code}]$$

$$\text{d2} = \dfrac{\dfrac{}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\}\ \rhd_\Psi l_{wJP} : \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp\ opnd}\,[\text{T-RegOp}]}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\}\ \mid pc \sqcup \tau(\mathsf{L})\ \rhd_\Psi\ \texttt{jmp}\ l_{wJP}[X]\ blk}\,[\text{T-Jmp}]$$

$$\dfrac{pc \sqcup \tau(\mathsf{L}) \sqcup \perp \sqsubseteq\ pc \qquad \dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\,[\text{ST-RegBank}]}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\} \mid pc \sqcup \tau(\mathsf{L}) \sqcup \perp\rangle^\perp}\,[\text{ST-Code}]$$

$$\text{d1} = \dfrac{d2 \qquad \dfrac{}{\Theta \mid \Gamma \rhd_\Psi r_a : \tau(\mathsf{L})\ opnd}\,[\text{T-RegOp}] \qquad \dfrac{}{\Theta \mid \Gamma \rhd_\Psi l_{bucle} : \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp}\,[\text{T-RegOp}]}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{L}), sp : l' \cdot \Sigma\}\ \mid pc\ \rhd_\Psi \texttt{bnz}\ r_a,\ l_{bucle}[X]; \texttt{jmp}\ l_{wJP}[X]\ blk}\,[\text{T-CondBranch}]$$

Luego por Lema 3 (b) podemos decir que: $\Theta \mid \Gamma\{sp : l' \cdot \Sigma\}\ \mid pc\ \rhd_\Psi |be|;\ \texttt{bnz}\ r_a,\ l_{bucle}[X]; \texttt{jmp}\ l_{wJP}[X]\ blk$

Para finalizar con el bloque de código l_{wJP}.

$$\dfrac{\perp \sqsubseteq\ pc \qquad \dfrac{0 \geq 0}{\rhd_\Psi \{sp : \Sigma\} \leq \{sp : \Sigma\}}\,[\text{T-RegBank}]}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\perp \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{\Gamma[sp := \Sigma]\} \mid \perp\rangle^\perp}$$

$$\dfrac{\Theta \rhd X\ \texttt{ok} \qquad \Gamma'(sp) = \Sigma \qquad \Gamma(sp) = l' \cdot \Sigma \qquad \Psi(l') = \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\perp}{\Theta \mid \Gamma\{sp : l' \cdot \Sigma\}\ \mid pc\ \rhd_\Psi \texttt{jmpJP}\ l'[X]\ blk}\,[\text{T-JmpJP}]$$

Caso T-WHILE para $\Delta(be) = \mathsf{H}$. Procedemos similar al caso anterior. Comenzaremos con el bloque l_0 con $\Psi(l_{fin}) = \Psi(l) = \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\perp$.

$$\dfrac{\dfrac{0 \geq 0}{\rhd_\Psi \{sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\,[\text{ST-RegBank}] \qquad pc \sqsubseteq\ pc}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\bot}\,[\text{ST-Code}]$$

$$\text{d1} = \dfrac{\dfrac{}{\Theta \mid \Gamma\{sp : l' \cdot \Sigma\}\ \rhd_\Psi l : \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid pc\rangle^\bot opnd}\,[\text{T-RegOp}]}{\Theta \mid \Gamma\{sp : l' \cdot \Sigma\}\ \mid pc\ \rhd_\Psi\ \texttt{jmp}\ l[X]\ blk}\,[\text{T-Jmp}]$$

$$\dfrac{d1 \qquad \Psi(l') = \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\bot \qquad pc \sqsubseteq pc \qquad \Gamma(sp) = \Sigma}{\Theta \mid \Gamma\{sp : \Sigma\}\ \mid pc\ \rhd_\Psi \texttt{pushJP}\ l'[X]; \texttt{jmp}\ l[X]\ blk}\,[\text{T-Push}]$$

Continuamos con el bloque l.

$$\dfrac{\dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\,[\text{ST-RegBank}] \qquad pc \sqcup \tau(\mathsf{H}) \sqsubseteq\ \top}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid \top\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \mid pc \sqcup \tau(\mathsf{H})\rangle^\bot}\,[\text{ST-Code}]$$

$$\text{d2} = \dfrac{\dfrac{}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\}\ \rhd_\Psi l_{wJP} : \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid \top\rangle^\bot opnd}\,[\text{T-RegOp}]}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\}\ \mid pc \sqcup \tau(\mathsf{H})\ \rhd_\Psi\ \texttt{jmp}\ l_{wJP}[X]\ blk}\,[\text{T-Jmp}]$$

$$\dfrac{pc \sqcup \tau(\mathsf{H}) \sqcup \bot \sqsubseteq\ \top \qquad \dfrac{1 \geq 0}{\rhd_\Psi \{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \leq \{sp : l' \cdot \Sigma\}}\,[\text{ST-RegBank}]}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid \top\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\} \mid pc \sqcup \tau(\mathsf{H}) \sqcup \bot\rangle^\bot}\,[\text{ST-Code}]$$

$$\text{d1} = \dfrac{d2 \qquad \dfrac{}{\Theta \mid \Gamma \rhd_\Psi r_a : \tau(\mathsf{H})\ opnd}\,[\text{T-RegOp}] \qquad \dfrac{}{\Theta \mid \Gamma \rhd_\Psi l_{bucle} : \texttt{CODE}\langle \forall[X]\{sp : l' \cdot \Sigma\} \mid \top\rangle^\bot}\,[\text{T-RegOp}]}{\Theta \mid \Gamma\{r_a : \tau(\mathsf{H}), sp : l' \cdot \Sigma\}\ \mid pc\ \rhd_\Psi \texttt{bnz}\ r_a,\ l_{bucle}[X]; \texttt{jmp}\ l_{wJP}[X]\ blk}\,[\text{T-CondBranch}]$$

Luego por Lema 3 (b) podemos decir que: $\Theta \mid \Gamma\{sp : l' \cdot \Sigma\}\ \mid pc\ \rhd_\Psi |be|;\ \texttt{bnz}\ r_a,\ l_{bucle}[X]; \texttt{jmp}\ l_{wJP}[X]\ blk$

Para finalizar con el bloque de código l_{wJP}.

$$\frac{\bot \sqsubseteq\ pc \qquad \dfrac{0 \geq 0}{\rhd_\Psi \{sp : \Sigma\} \leq \{sp : \Sigma\}}\ \text{[T-RegBank]}}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{\Gamma[sp := \Sigma]\} \mid \bot\rangle^\bot}$$

$$\frac{\Theta \rhd X\ \texttt{ok} \qquad \Gamma'(sp) = \Sigma \qquad \Gamma(sp) = l' \cdot \Sigma \qquad \Psi(l') = \texttt{CODE}\langle \forall[X]\{sp : \Sigma\} \mid pc\rangle^\bot}{\Theta \mid \Gamma\{sp : l' \cdot \Sigma\}\ \mid \top\ \rhd_\Psi \texttt{jmpJP}\ l'[X]\ blk}\ \text{[T-JmpJP]}$$

Caso T-STM. Por H.I. sobre stm_1, podemos decir que $\rhd H_1 : \Psi_1$. Luego, por H.I. sobre stm_2 concluimos diciendo que $\rhd H_2 : \Psi_2$.

Caso T-Prog. La traducción inicial de un programa en WHILE, genera dos bloques de código nuevos, l_{main} y l_{halt}. Vamos a comenzar por l_{main} con $\Psi(l_{fin}) = \Psi(l) = \texttt{CODE}\langle \forall[X]\{sp : Halt.X\} \mid \bot\rangle^\bot$.

$$\frac{\dfrac{0 \geq 0}{\rhd_\Psi \{sp : Halt.X\} \leq \{sp : Halt.X\}}\ \text{[ST-RegBank]} \qquad \bot \sqsubseteq\ \bot}{\rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : Halt.X\} \mid \bot\rangle^\bot \leq \rhd_\Psi \texttt{CODE}\langle \forall[X]\{sp : Halt.X\} \mid \bot\rangle^\bot}\ \text{[ST-Code]}$$

$$\frac{\dfrac{}{\Theta \mid \Gamma\{sp : Halt.X\}\ \rhd_\Psi l : \texttt{CODE}\langle \forall[X]\{sp : Halt.X\} \mid \bot\rangle^\bot\ opnd}\ \text{[T-RegOp]}}{\Theta \mid \Gamma\{sp : Halt.X\}\ \mid \bot\ \rhd_\Psi \texttt{jmp}\ l[X]\ blk}\ \text{[T-Jmp]}$$

Finalmente, l_{halt}:

$$\frac{\Gamma(sp) = Halt.X \qquad \Theta \rhd \Gamma\ \texttt{ok}}{\Theta \mid \Gamma\{sp : Halt.X\}\ \mid \bot\ \rhd_\Psi \texttt{halt}\ blk}\ \text{[T-Halt]}$$

□

Teorema 3 (traducción de While). Si $\Delta \sim \Gamma(sp)$, $\Delta \vdash stm$, $\rhd H_0 : \Psi_0$,

$$\left|\frac{d}{\Delta \vdash stm}\right| \begin{bmatrix} \Psi_0 \\ H_0 \end{bmatrix} = \begin{bmatrix} \Psi \\ H \\ X \end{bmatrix}, \text{ entonces, } \rhd (H, \{sp : Halt.X\}, \texttt{jmp}\ l_{main}[X]).$$

Demostración. Finalmente, la demostración del Teorema 3 se obtiene dado que por Lema 4 sabemos que $\triangleright H : \Psi$, que incluye el bloque de código inicial l_{main}. Dado que $\triangleright_{\Psi} R : \Gamma$ regBank, podemos decir que: $\triangleright(H, R \cup \{sp\}, B) : [\Psi, \Gamma \cup \{sp : Halt.X\}, \bot]$ machConfig □

Capítulo 4

Implementación

Como parte de este trabajo se implementó el compilador desarrollado en el capítulo 3, el cual traduce programas escritos en While a programas en SecTAL. También, se implementó un chequeador de tipos para SecTAL[1]. Dicha implementación permitió explorar lo expuesto anteriormente desde el punto de vista práctico. Mostrando tanto que la compilación con preservación de tipos es aplicable para lenguajes que expresan propiedades de seguridad no tradicionales como no-interferencia como la comprobación de la utilidad del lenguaje SecTAL como lenguaje objeto de este tipo de compiladores.

La implementación del compilador y del chequeador de tipos se realizó en el lenguaje Java versión 1.4.1. Para integrarlos en una única aplicación se implementó un front-end en Java Swing que permite escribir programas en While, compilarlos y correr el chequeador de tipos de SecTAL sobre el resultado de la compilación. También permite escribir directamente programas en SecTAL.

Este capítulo presenta una descripción general de la implementación del compilador y del chequeador de tipos. El código fuente de ambos es de libre acceso[2].

4.1. Implementación de los Lenguajes

Tanto el compilador de While como el chequeador de tipos de SecTAL se implementaron utilizando SableCC [GAG98]. SableCC es un framework orientado a objetos para generar parsers del tipo LALR(1) a diferencia de JavaCC de Sun Microsystems que genera parsers del tipo LL(k). SableCC genera el front-end de un compilador, es decir, un analizar léxico y sintáctico a partir de un archivo de entrada donde se especifican las reglas léxicas y sintácticas. Más detalles sobre los algoritmos de parsing LL(k) y LALR(1), y las herramientas de generación de parsers SableCC y JavaCC pueden

[1] Si bien el compilador de While incluye su propio chequeador de tipos, siempre que hagamos referencia a un chequeador de tipos estaremos hablando del de SecTAL.

[2] Para ver la aplicación o descargar los fuentes se puede consultar `www.lifia.info.unlp.edu.ar/~eduardo/compilacionSegura/index.html`.

encontrarse en [APP02].

4.1.1. El Compilador de While

La Figura 4.1 presenta un diagrama con las fases del compilador de While. La primera fase utiliza el generador de parsers SableCC para generar el front-end del compilador. Las tres últimas fases implementan el back-end del compilador. A continuación, se explican cada una de estas fases.

1. Análisis léxico y sintáctico

    1. Se escribe el archivo con la especificación requerida por SableCC que contiene expresiones regulares para definir los componentes léxicos (tokens) y la gramática para definir la sintaxis de While.
    2. En este paso se ejecuta SableCC utilizando como entrada la especificación definida en el paso anterior.
    3. Luego de ejecutar SableCC, como salida se generan las clases Java que conforman el front-end del compilador. Dichas clases forman un framework de caja blanca que incluye el analizador léxico, el analizador sintáctico y el árbol de sintaxis abstracto (en adelante AST). Mediante el mecanismo de herencia, podemos instanciar este framework para implementar las fases siguientes del compilador. La estructura genérica de control del framework generado por SableCC es el recorrido que hace del AST, realizando llamadas a las subclases (instanciadoras del framework) a medida que reconoce sentencias del lenguaje.

2. Este paso y los siguientes, son pasadas que recorren el AST para implementar el back-end del compilador. En esta primera pasada generamos la tabla de símbolos para manejar las variables y sus propiedades: el ámbito y su tipo L o H.

3. La segunda pasada realiza el chequeo de tipos, implementando las reglas de tipado definidas en las Figuras 3.2 y 3.3.

4. La tercera y última pasada genera código y tipos de SecTAL.

4.1.2. El Chequeador de Tipos de SecTAL

Tal como se menciona al principio de esta sección, también utilizamos SableCC para la implementación del chequeador de tipos de SecTAL. Los pasos explicados anteriormente y presentados en la Figura 4.1, existen también aquí, con excepción de la fase de generación de código. Luego de realizado el análisis léxico y sintáctico del programa SecTAL, el chequeador de tipos indica si el programa contiene o no flujos de información inválidos.

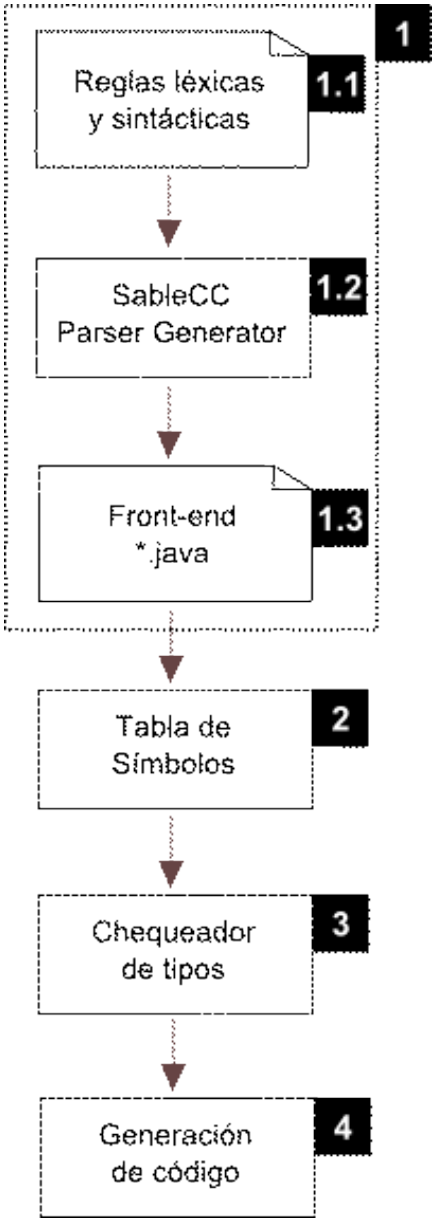

Figura 4.1: Fases del Compilador de WHILE

El chequeador de tipos implementa el conjunto de reglas de tipado presentadas en la sección 2.4. Cada regla de tipado es un *objeto* que implementa una *interfaz* y se construye con las precondiciones de la regla. Como ejemplo, a continuación se muestra la *interfaz* TypingRule y la implementación de la regla de tipado [T-Mov], MovRule.

```
public interface TypingRule {
    public void evaluate() throws TypeCheckerException;
}

public class MovRule implements TypingRule {
    private Register  destination;
    private WordValue source;

    public MovRule(Register destination, WordValue source) {
        this.destination = destination;
        this.source      = source;
    }
    public void evaluate() throws TypeCheckerException {
        //rule implementation
    }
}
```

En caso de que el chequeador de tipos detecte flujos de información inválidos, también puede indicar cual fue la regla de tipado que falló.

4.2. Testing

La implementación también incluye utilidades y scripts para realizar testing automatizado, tanto para el chequeador de tipos de SecTAL como para el compilador de While.

A lo largo del desarrollo del chequeador de tipos, se escribieron varios programas en SecTAL, que luego eran utilizados para probar su funcionamiento. Dentro de este conjunto de programas de prueba, existían los que debían pasar el chequeo de tipos como los que no debían pasar, es decir, aquellos que contenían flujos de información inválidos. Para aquellos programas que contenían flujos de información inválidos, el chequeador de tipos debía contestarnos con un TypeCheckerException y para los que contenían flujos de información válidos, finalizar correctamente. Para esto, se desarrolló una utilidad para usar en conjunto con JUnit[3], que permite escribir casos de prueba automatizados. Como ejemplo, a continuación se muestra una clase de test del chequeador de tipos:

```
public class TestTypeChecker extends TestCase {
```

[3] http://junit.org/

```
        public void testMoveRule() {
            UtilsTest.typeChecked("movRuleTypeCheckedProgram.sectal");
        }

        public void testArithRule() {
            UtilsTest.notTypeChecked("arithRuleNotTypeCheckedProgram.sectal");
        }
    }
```

En este ejemplo presentado, se testean dos programas SecTAL, uno que contiene flujos de información válidos (*movRuleTypeCheckedProgram.sectal*) y otro que tiene flujos de información inválidos (*arithRuleNotTypeCheckedProgram.sectal*). Los métodos UtilsTest.typeChecked y UtilsTest.notTypeChecked, contienen los *assertions* correspondientes, es decir, si la ejecución del test testMoveRule(), genera una TypeCheckerException, entonces el test fallará. Si la ejecución del test testArithRule() no genera una TypeCheckerException, entonces el test fallará.

Para testear el compilador, sería apropiado compilar un conjunto de programas en While y ejecutar el chequeador de tipos de SecTAL sobre la salida del compilador. Para ello, creamos un script que toma como entrada un conjunto de programas escritos en While, compila cada uno de estos programas y corre el chequeador de tipos de SecTAL sobre la salida. Todos los programas While, sintácticamente correctos y que pasan el chequeo de tipos, deben generar código SecTAL seguro, es decir, código SecTAL que pase el chequeo de tipos.

Los test automatizados producidos durante la implementación son importantes a la hora de extender el compilador de While para que sean utilizados como test de regresión.

4.3. Interfaz Gráfica

Como se mencionó al comienzo de este capítulo se implementó un front-end en Java Swing, para integrar el compilador de While y el chequeador de tipos de SecTAL en una única aplicación.

La Figura 4.2 presenta una captura de pantalla de la interfaz gráfica, inmediatamente después de compilar el programa en While, presentando en el ejemplo de la sección 3.2.3.

La interfaz cuenta con tres paneles. Sobre el panel de la izquierda se pueden escribir programas en While o seleccionar uno predefinido de la lista desplegable. En el panel de la derecha, se puede realizar los mismo pero sobre el lenguaje SecTAL. Además, cuando se compila un programa While, el resultado de la compilación es almacenado en éste panel. Luego, es posible ejecutar el chequeador de tipos de SecTAL.

Finalmente, el panel inferior es donde se escriben los errores que pueden generar

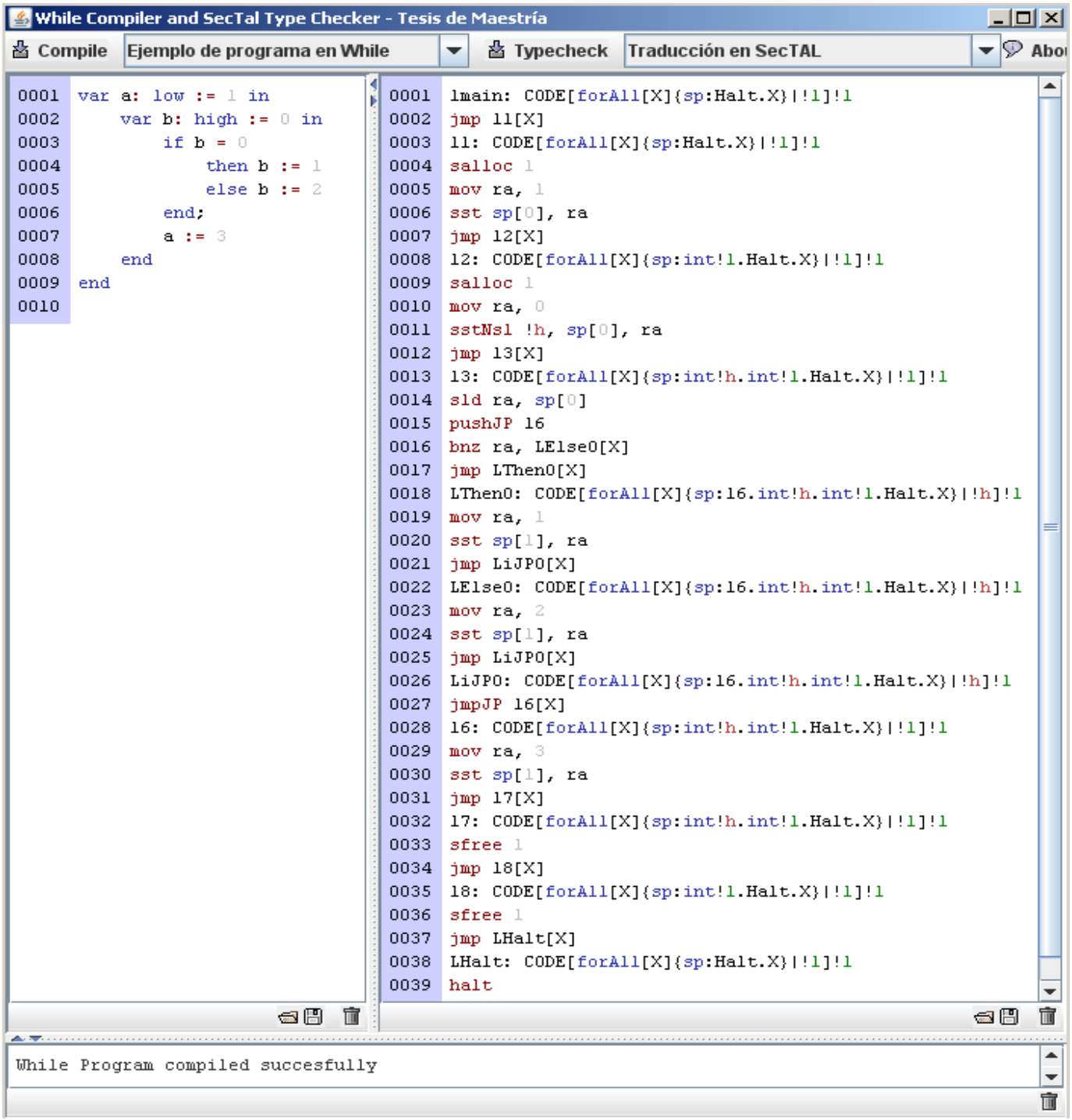

Figura 4.2: Interfaz Gráfica de la Aplicación

el compilador de While o el chequeador de tipos SecTAL. Los mensajes de error también informan la línea de código, tanto de While como de SecTAL donde el error se produjo.

Capítulo 5

Conclusiones

En este capítulo presentamos las conclusiones, resumimos las contribuciones y realizamos algunas propuestas como trabajo futuro.

5.1. Resumen, Conclusiones y Contribuciones

En este trabajo se presentó un lenguaje de alto nivel con control de flujo de información llamado While, e incluimos una función de compilación que traduce programas escritos en While a programas escritos en SecTAL. También se ha presentando una demostración que muestra que la traducción preserva tipos, en el sentido de que mapea programas bien tipados en While a programas bien tipados en SecTAL.

Resumimos las contribuciones de la siguiente manera:

- La definición de una función de compilación de un lenguaje imperativo sencillo hacia un lenguaje de bajo nivel basado en Typed Assembly Language.
- La prueba de un resultado de preservación de tipado que muestra que si el programa fuente es bien tipado (y por ende seguro) también lo será el resultado de compilar el mismo.

La función de compilación ha sido implementada y también un chequeador de tipos de SecTAL. De esta forma mostramos que la compilación con preservación de tipos es aplicable para lenguajes que expresan propiedades de seguridad no tradicionales como no-interferencia y a su vez la utilidad del lenguaje SecTal como lenguaje objeto de este tipo de compiladores. Ricardo Medel menciona en su tesis de doctorado [MED06], como parte de la sección de trabajo futuro, los temas presentados en esta tesis. A saber, la realización de una función de compilación junto con una prueba de preservación de tipos, y la implementación de dicha función más la implementación de un chequeador de tipos de SIFTAL.

Durante la definición de la función de compilación detectamos que era necesario agregar una instrucción en SecTAL, no presente en SIFTAL. Dicha instrucción, que llamamos `sstNsl` $level,\ sp[i],\ r_s$, y se explica en los capítulos 2 y 3, merece una mención extra aquí.

Greg Morrisett [MOR85], en su tesis de doctorado, define brevemente el significado de **preservar tipos en compilación** de la siguiente forma:

1. El lenguaje fuente y el lenguaje objeto deben ser tipados.
2. Se debe proveer tanto la función para traducir sentencias y expresiones como para traducir tipos.
3. Si una expresión e tiene tipo t, la compilación de e es e' y la compilación de t es t', entonces e' tiene tipo t'.

Basándonos en esta explicación, al inicio de la sección 3.2 definimos la función τ que mapea tipos de While a SecTAL. De esta forma, la correspondencia de tipos en While con tipos de SecTAL se define como: $\Delta \sim \Gamma(sp)$. Detectamos que esta correspondencia no era posible de satisfacer para la traducción de la regla $[\mathsf{T} - \mathsf{DecHigh}]$ con la actual definición de SIFTAL [BCM04, MED06]. Para resolver esta cuestión fue necesario introducir la instrucción `sstNsl` $level,\ sp[i],\ r_s$. Dicha instrucción guarda en la pila el registro r_s con tipo $level$ independientemente del tipo actual de r_s. Se puede observar en la traducción de la regla $[\mathsf{T} - \mathsf{DecHigh}]$, explicada en la sección 3.2.2, que para este caso $level = \tau(\mathsf{H})$.
La alternativa era utilizar la instrucción `sst` $sp[i],\ r_s$, pero en caso de que la traducción de la expresión $|e|$ deje el registro r_s con tipo $\tau(\mathsf{L})$, estaríamos guardando en la pila el tipo $\tau(\mathsf{L})$. Dado que por la traducción de la regla $[\mathsf{T} - \mathsf{DecHigh}]$, estamos declarando una variable con tipo H en While, la definición $\Delta \sim \Gamma(sp)$ no se estaría respetando.

5.2. Trabajo Futuro

En esta sección realizamos algunas propuestas con la intención de ampliar y mejorar el trabajo.

Primero, sería pertinente ampliar este trabajo, aumentando el poder expresivo del lenguaje While. El paso inmediato sería agregarle funciones. Esto no representaría mayor esfuerzo, dado que en la función de compilación presentada las variables son almacenadas en la pila de SecTAL. Este mismo mecanismo, tanto para la nueva regla de traducción como para su prueba se utilizaría para el pasaje de parámetros.
Luego, ir incorporando diferentes construcciones sintácticas para terminar transformando el lenguaje While en un subconjunto de un lenguaje utilizado en la industria como podría ser Java. Nuestra intención no es realizar un trabajo similar a JFlow [MYE99]

(al día de la fecha no cuenta con la demostración de la correctitud de su sistema), sino, incrementar el poder expresivo de nuestro lenguaje fuente manteniendo sus propiedades de seguridad, y más importante aun, manteniendo la corrección de su traducción a SecTAL.

En segundo lugar, otra mejora importante sería enriquecer el lenguaje While para permitir n niveles de seguridad. Para esto es necesario incorporar a la sintaxis las estructuras léxicas que permitan al programador escribir y definir los niveles de seguridad a utilizar y el orden entre ellos. También sería necesario modificar el sistema de tipos, con la posibilidad de definirlo de forma similar al presentado en [VSI96]. Por último, habría que adaptar la función de compilación y la prueba de preservación de tipos.

Por otro lado, para que los lenguajes con control de flujo de información basados en la verificación de no-interferencia sean utilizados en la práctica, es necesario desclasificar cierta información. En muchos casos no-interferencia es muy estricto. El ejemplo clásico es el programa para verificar contraseñas. El sólo hecho de informar que una contraseña es incorrecta genera un flujo de información inválido, rechazando el programa como inseguro. Por este motivo sería de interés incorporar mecanismos de desclasificación [SS05].

Finalmente, por el lado de SecTAL es necesario agregar más instrucciones de salto condicional para poder traducir expresiones booleanas más ricas. Actualmente solo cuenta con `bnz` r, v (salta si r $\neq 0$). Sería ideal incorporar aquellas definidas para STAL [MCGW02]. A saber, `beq` r, v (salta si r $= 0$), `bgt` r, v (salta si r > 0), `blt` r, v (salta si r < 0), `bgte` r, v (salta si r ≥ 0), `blte` r, v (salta si r ≤ 0). Note que agregar dichas instrucciones no altera la demostración de correctitud del sistema.

Apéndice A

No-Interferencia en While

En este sección realizaremos la demostración de los lemas y del teorema de no-interferencia para el lenguaje While. A continuación repetimos la definición de la relación de equivalencia, y luego introduciremos una serie de lemas que utilizaremos para la demostración del Teorema 2.

Definición 1 (equivalencia de estados). Dados dos estados σ_1 y σ_2 y un contexto de tipado Δ, $\sigma_1 \equiv_L^\Delta \sigma_2 \Leftrightarrow \forall x \in \text{dom}(\Delta)$, $\Delta(x) = \mathsf{L} \Rightarrow \sigma_1(x) = \sigma_2(x)$. La relación $\equiv_L^\Delta$ es de equivalencia.

Lema 2 (relación de equivalencia). La relación $\equiv_L^\Delta$ es de equivalencia.

Lema A.1 (variables públicas). Si $\Delta \vdash e : \mathsf{L}$ entonces, $\forall x$ en $Vars(e)$, $\Delta(x) = \mathsf{L}$. Donde $Vars(e)$ es una función que devuelve el conjunto de variables involucradas en e. Lo mismo aplica para las expresiones booleanas *be*.

Demostración. La demostración es por inducción estructural en e y *be*:

Caso $e = i$**.** Dado que no hay variables involucradas en e concluimos.

Caso $e = x$**.** Si $\Delta \vdash x : \mathsf{L}$, entonces por regla [T-EVar] sabemos que $\Delta(x) = \mathsf{L}$.

Caso $e = e_1 + e_2$**.** Si $\Delta \vdash e_1 + e_2 : \mathsf{L}$, entonces por regla [T-ESuma] sabemos que $t_1 \sqcup t_2 = \mathsf{L}$. Por la definición de *junta* sabemos que $t_1 = \mathsf{L}$ y $t_2 = \mathsf{L}$. Lo cual significa que $\Delta \vdash e_1 : \mathsf{L}$ y $\Delta \vdash e_2 : \mathsf{L}$. Dado que e_i (para i = 1,2) es una sub-expresión inmediata de $e_1 + e_2$ y $Vars(e_i) \subseteq Vars(e_1 + e_2)$ podemos aplicar la H.I sobre e_i y concluir.

Caso $be = true$**.** Dado que no hay variables involucradas en *be* concluimos.

Caso $be = false$**.** Idem caso anterior.

Caso $be = e_1 = 0$**.** Si $\Delta \vdash e_1 = 0 : \mathsf{L}$, entonces por regla [T-BEIgual] sabemos que $\Delta \vdash e_1 : \mathsf{L}$. Luego concluimos por H.I.

□

Lema A.2 (expresiones públicas). Si $\Delta \vdash e : \mathsf{L}$ y $\sigma_1 \equiv_L^\Delta \sigma_2$, entonces $\mathfrak{A}[e]\sigma_1 = \mathfrak{A}[e]\sigma_2$. Si $\Delta \vdash be : \mathsf{L}$ y $\sigma_1 \equiv_L^\Delta \sigma_2$, entonces $\mathfrak{B}[be]\sigma_1 = \mathfrak{B}[be]\sigma_2$.

Demostración. La demostración es por inducción estructural en e y be:

Caso $e = i$**.** Por definición de la función $\mathfrak{A}$, sabemos que: $\mathfrak{A}[i]\sigma_1 = \mathfrak{A}[i]\sigma_2 = i$

Caso $e = x$**.** Por definición de la función $\mathfrak{A}$, sabemos que: $\mathfrak{A}[x]\sigma_1 = \sigma_1(x)$ y $\mathfrak{A}[x]\sigma_2 = \sigma_2(x)$. Si $\Delta \vdash e : \mathsf{L}$, entonces por Lema A.1 $\Delta(x) = \mathsf{L}$. Sabiendo que $\sigma_1 \equiv_L^\Delta \sigma_2$, podemos decir que $\sigma_1(x) = \sigma_2(x)$, lo cual significa que $\mathfrak{A}[x]\sigma_1 = \mathfrak{A}[x]\sigma_2$.

Caso $e = e_1 + e_2$**.** Por definición de $\mathfrak{A}$, sabemos que: $\mathfrak{A}[e_1 + e_2]\sigma_1 = \mathfrak{A}[e_1]\sigma_1 + \mathfrak{A}[e_2]\sigma_1$ y $\mathfrak{A}[e_1 + e_2]\sigma_2 = \mathfrak{A}[e_1]\sigma_2 + \mathfrak{A}[e_2]\sigma_2$. Por H.I. decimos que $\mathfrak{A}[e_1]\sigma_1 = \mathfrak{A}[e_1]\sigma_2$ y que $\mathfrak{A}[e_2]\sigma_1 = \mathfrak{A}[e_2]\sigma_2$. Luego concluimos con $\mathfrak{A}[e_1]\sigma_1 + \mathfrak{A}[e_2]\sigma_1 = \mathfrak{A}[e_1]\sigma_2 + \mathfrak{A}[e_2]\sigma_2$.

Caso $be = true \mid false$**.** Por definición de $\mathfrak{B}$ sabemos que $\mathfrak{B}[true]\sigma_1 = \mathfrak{B}[true]\sigma_2 = true$. Lo mismo ocurre con $false$.

Caso $be = e_1 = 0$**.** Por definición de $\mathfrak{B}$ sabemos que: $\mathfrak{B}[e_1 = 0]\sigma_1 = true$ si $\mathfrak{A}[e_1]\sigma_1 = 0$, $false$ en otro caso. Y $\mathfrak{B}[e_1 = 0]\sigma_2 = true$ si $\mathfrak{A}[e_1]\sigma_2 = 0$, $false$ en otro caso. Por H.I. podemos decir que $\mathfrak{A}[e_1]\sigma_1 = \mathfrak{A}[e_1]\sigma_2$ lo que significa que $\mathfrak{B}[e_1 = 0]\sigma_1 = \mathfrak{B}[e_1 = 0]\sigma_2$.

□

Lema A.3 (sentencias en contexto privado). Las sentencias que son tipadas en un contexto de seguridad H no modifican valores de variables L. Si $\Delta, \mathsf{H} \vdash stm$ y $\langle stm, \sigma_1\rangle \Downarrow \sigma_2$, entonces $\sigma_1 \equiv_L^\Delta \sigma_2$.

Demostración. La demostración es por inducción sobre la estructura del árbol de derivación de $\langle stm, \sigma_1\rangle \Downarrow \sigma_2$. Primero demostraremos que esta propiedad se cumple para los axiomas del sistema de transición. Luego, asumiendo que la propiedad se cumple para las premisas de las reglas (hipótesis inductiva), probaremos que se cumple para la conclusión. En todos los casos asumimos que la raíz del árbol de derivación es la transición $\langle stm, \sigma_1\rangle \Downarrow \sigma_2$ donde stm es instanciada por el caso a ser analizado. Asumimos que la ejecución de $\langle stm, \sigma_1\rangle \Downarrow \sigma_2$ termina.

Caso E-Asig. Tenemos que: $\langle x := e, \sigma_1\rangle \Downarrow \sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1]$. El sistema de tipos requiere que $\Delta(x) = \mathsf{H}$ para tipar la sentencia $x := e$ con contexto de seguridad H. Dado que no se modifican variables con tipo L, podemos decir que $\sigma_1 \equiv_L^\Delta \sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1]$.

Caso E-IFTrue/E-IFFalse. Dado que el contexto de tipado es H, por regla [T-IF] sabemos que $\Delta \vdash be : \mathsf{H}$, $\Delta, \mathsf{H} \vdash stm_1$ y $\Delta, \mathsf{H} \vdash stm_2$. Dependiendo del resultado de $\mathfrak{B}[be]\sigma_1$, se ejecutará stm_1 o stm_2. Por H.I. podemos decir que para $\langle stm_1, \sigma_1\rangle \Downarrow \sigma_2, \sigma_1 \equiv_L^\Delta \sigma_2$ y para $\langle stm_2, \sigma_1\rangle \Downarrow \sigma_2, \sigma_1 \equiv_L^\Delta \sigma_2$. Independientemente de la evaluación de la condición, obtenemos $\sigma_1 \equiv_L^\Delta \sigma_2$.

Caso E-Decl. Dado que el contexto de tipado es H, por regla [T-DecHigh] sabemos que $\Delta(x) = \mathsf{H}$ y $\Delta, \mathsf{H} \vdash stm$. Dado que no se modifican variables con tipo L, podemos decir que $\sigma_1 \equiv_L^\Delta \sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1]$. Por H.I. podemos decir que $\sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1] \equiv_L^\Delta \sigma_2$ y por transitividad que $\sigma_1 \equiv_L^\Delta \sigma_2 \equiv_L^\Delta \sigma_2[x \mapsto \sigma_1(x)]$.

Caso E-WHILETrue/E-WHILEFalse. Dado que el contexto de tipado es H, por regla [T-WHILE] sabemos que $\Delta \vdash be : \mathsf{H}$ y $\Delta, \mathsf{H} \vdash stm_1$. Si $\mathfrak{B}[be]\sigma_1 = false$, concluimos. Si $\mathfrak{B}[be]\sigma_1 = true$, tenemos $\langle stm_1, \sigma_1\rangle \Downarrow \sigma_1'$ y $\langle while\ be\ do\ stm_1\ end, \sigma_1'\rangle \Downarrow \sigma_2$. Por H.I. tenemos que $\sigma_1 \equiv_L^\Delta \sigma_1'$ y $\sigma_1' \equiv_L^\Delta \sigma_2$, luego por transitividad $\sigma_1 \equiv_L^\Delta \sigma_2$.

Caso E-Stm. Dado que el contexto de tipado es H, por regla [T-Stm] sabemos que $\Delta, \mathsf{H} \vdash stm_1$ y $\Delta, \mathsf{H} \vdash stm_2$. Luego tenemos $\langle stm_1, \sigma_1\rangle \Downarrow \sigma_1'$ y $\langle stm_2, \sigma_1'\rangle \Downarrow \sigma_2$. Por H.I. tenemos que $\sigma_1 \equiv_L^\Delta \sigma_1'$ y $\sigma_1' \equiv_L^\Delta \sigma_2$, luego por transitividad $\sigma_1 \equiv_L^\Delta \sigma_2$.

□

Definición 2 (programas no-interferentes). Un programa escrito en WHILE es no-interferente, denotado así $\mathrm{NI}_\Delta(\mathrm{stm})$, si dados los estados $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ donde $\sigma_1 \equiv_L^\Delta \sigma_2$, si $\langle stm, \sigma_1\rangle \Downarrow \sigma_3$ y $\langle stm, \sigma_2\rangle \Downarrow \sigma_4$ implica $\sigma_3 \equiv_L^\Delta \sigma_4$.

Teorema 2 (no-interferencia). Dado un programa stm escrito en WHILE, si $\Delta \vdash stm$ implica $\mathrm{NI}_\Delta(\mathrm{stm})$.

Demostración. La demostración es por inducción sobre la estructura del árbol de derivación de $\langle stm, \sigma_1\rangle \Downarrow \sigma_3$. Nuevamente, asumimos que las ejecuciones terminan.

Caso E-Asig. Tenemos que: $\langle x := e, \sigma_1\rangle \Downarrow \sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1]$, y $\langle x := e, \sigma_2\rangle \Downarrow \sigma_2[x \mapsto \mathfrak{A}[e]\sigma_2]$.

Si $\Delta(x) = \mathsf{H}$, tenemos que $\sigma_1 \equiv_L^\Delta \sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1]$ y $\sigma_2 \equiv_L^\Delta \sigma_2[x \mapsto \mathfrak{A}[e]\sigma_2]$. Luego, dado que $\sigma_1 \equiv_L^\Delta \sigma_2]$ podemos decir que $\sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1] \equiv_L^\Delta \sigma_2[x \mapsto \mathfrak{A}[e]\sigma_2]$.

Si $\Delta(x) = \mathsf{L}$, sabemos por regla [T-AsigLow] que $\Delta \vdash e : \mathsf{L}$. Luego, por Lema A.2 tenemos que $\mathfrak{A}[e]\sigma_1 = \mathfrak{A}[e]\sigma_2$. Con lo cual, sabiendo que $\sigma_1 \equiv_L^\Delta \sigma_2$, podemos decir que $\sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1] \equiv_L^\Delta \sigma_2[x \mapsto \mathfrak{A}[e]\sigma_2]$.

Caso E-IFTrue/E-IFFalse. Dos casos son posibles:

Si $\Delta \vdash be : \mathsf{L}$, por Lema A.2 podemos decir que $\mathfrak{B}[be]\sigma_1 = \mathfrak{B}[be]\sigma_2$. Esto significa que ambas derivaciones tomarán el mismo camino de ejecución. Entonces,

suponiendo que $\mathfrak{B}[be]\sigma_1 = \text{true}$, tenemos que $\langle stm_1, \sigma_1\rangle \Downarrow \sigma_3$ y $\langle stm_1, \sigma_2\rangle \Downarrow \sigma_4$. Aplicando H.I. tenemos que $\sigma_3 \equiv_L^\Delta \sigma_4$. El caso si $\mathfrak{B}[be]\sigma_1 = \text{false}$ es análogo.

Si $\Delta \vdash be : \mathsf{H}$, el sistema de tipos requiere que el contexto de seguridad sea H. Luego, por Lema A.3 podemos decir que $\sigma_1 \equiv_L^\Delta \sigma_3$ y $\sigma_2 \equiv_L^\Delta \sigma_4$, con lo cual, $\sigma_3 \equiv_L^\Delta \sigma_4$.

Caso E-Decl. Dos casos son posibles:

Si $\Delta(x) = \mathsf{L}$, sabemos por regla [T-AsigLow] que $\Delta \vdash e : \mathsf{L}$. Luego, por Lema A.2 tenemos que $\mathfrak{A}[e]\sigma_1 = \mathfrak{A}[e]\sigma_2$, con lo cual podemos decir que $\sigma_1 \equiv_L^\Delta \sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1]$ y $\sigma_2 \equiv_L^\Delta \sigma_2[x \mapsto \mathfrak{A}[e]\sigma_2]$, luego $\sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1] \equiv_L^\Delta \sigma_2[x \mapsto \mathfrak{A}[e]\sigma_2]$. Aplicando H.I. tenemos que $\sigma_3 \equiv_L^\Delta \sigma_4$.

Si $\Delta(x) = \mathsf{H}$, tenemos que $\sigma_1 \equiv_L^\Delta \sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1]$ y $\sigma_2 \equiv_L^\Delta \sigma_2[x \mapsto \mathfrak{A}[e]\sigma_2]$, luego $\sigma_1[x \mapsto \mathfrak{A}[e]\sigma_1] \equiv_L^\Delta \sigma_2[x \mapsto \mathfrak{A}[e]\sigma_2]$. Aplicando H.I. tenemos que $\sigma_3 \equiv_L^\Delta \sigma_4$.

Caso E-WHILETrue/E-WHILEFalse. Dos casos son posibles:

Si $\Delta \vdash be : \mathsf{L}$, por Lema A.2 podemos decir que $\mathfrak{B}[be]\sigma_1 = \mathfrak{B}[be]\sigma_2$. Esto significa que ambas derivaciones tomarán el mismo camino de ejecución. Entonces, suponiendo que $\mathfrak{B}[be]\sigma_1 = \text{true}$, tenemos $\langle stm_1, \sigma_1\rangle \Downarrow \sigma_1'$, $\langle \textit{while } be \textit{ do } stm_1 \textit{ end}, \sigma_1'\rangle \Downarrow \sigma_3$ y $\langle stm_1, \sigma_2\rangle \Downarrow \sigma_2'$ y $\langle \textit{while } be \textit{ do } stm_1 \textit{ end}, \sigma_2'\rangle \Downarrow \sigma_4$. Aplicando H.I. sobre $\langle stm_1, \sigma_1\rangle \Downarrow \sigma_1'$ y $\langle stm_1, \sigma_2\rangle \Downarrow \sigma_2'$, tenemos que $\sigma_1' \equiv_L^\Delta \sigma_2'$. Luego, aplicando H.I. sobre $\langle \textit{while } be \textit{ do } stm_1 \textit{ end}, \sigma_1'\rangle \Downarrow \sigma_3$ y $\langle \textit{while } be \textit{ do } stm_1 \textit{ end}, \sigma_2'\rangle \Downarrow \sigma_4$, tenemos que $\sigma_3 \equiv_L^\Delta \sigma_4$. Si $\mathfrak{B}[be]\sigma_1 = \text{false}$, concluimos.

Si $\Delta \vdash be : \mathsf{H}$, el sistema de tipos requiere que el contexto de seguridad sea H. Luego, por Lema A.3 podemos decir que $\sigma_1 \equiv_L^\Delta \sigma_3$ y $\sigma_2 \equiv_L^\Delta \sigma_4$, con lo cual, $\sigma_3 \equiv_L^\Delta \sigma_4$.

Caso E-Stm. Tenemos derivaciones para $\langle stm_1, \sigma_1\rangle \Downarrow \sigma_1'$, $\langle stm_2, \sigma_1'\rangle \Downarrow \sigma_3$ y $\langle stm_1, \sigma_2\rangle \Downarrow \sigma_2'$, $\langle stm_2, \sigma_2'\rangle \Downarrow \sigma_4$. Aplicando H.I. sobre $\langle stm_1, \sigma_1\rangle \Downarrow \sigma_1'$ y $\langle stm_1, \sigma_2\rangle \Downarrow \sigma_2'$ tenemos que $\sigma_1' \equiv_L^\Delta \sigma_2'$. Luego, aplicando H.I. sobre $\langle stm_2, \sigma_1'\rangle \Downarrow \sigma_3$ y $\langle stm_2, \sigma_2'\rangle \Downarrow \sigma_4$ tenemos que $\sigma_3 \equiv_L^\Delta \sigma_4$.

□

Bibliografía

[AGA00] J. Agat, Transforming out timing leaks. POPL 2000: 40-53.

[APP02] W. Appel, *Modern compiler implementation in Java.* Cambridge University Press, 2002. Segunda Edición.

[BB08] F. Bavera, E. Bonelli, Type-based information flow analysis for bytecode languages with variable object field policies. SAC 2008: 347-351

[BBR04] G. Barthe, A. Basu, and T. Rezk, Security types preserving compilation. VMCAI 2004: 2-15.

[BN02] A. Banerjee and D. A. Naumann, Secure information flow and pointer confinement in a java-like language. CSFW 2002: 253.

[BP73] D. E. Bell and L. J. La Padula, Secure computer systems: Mathematical foundations. Technical Report MTR-2547 Vol. I, The MITRE Corporation, Bedford, Massachusetts, March 1973.

[BCM04] E. Bonelli, A. Compagnoni, and R. Medel, Information Flow Analysis for a Typed Assembly Language with Polymorphic Stacks. CASSIS 2005: 37-56

[EBM09] E. Bonelli, E. Molinari, Compilación de Programas Seguros. ASSE, Simposio Argentino de Ingeniería de Software, 38 JAIIO 2009.

[DD77] Dorothy E. Denning and Peter J. Denning, Certification of programs for secure information flow. Commun. ACM 20(7): 504-513 (1977).

[DEN76] Dorothy E. Denning, A lattice model of secure information flow. Commun. ACM 19(5): 236-243 (1976).

[FL91] C. Fischer and R. LeBlanc, *Crafting a Compiler with C.* Addison Wesley, 1991. Primera Edición.

[GAG98] E. Gagnon, SableCC, an object oriented compiler framework. Master's thesis, McGill University, March 1998.

[GJS96] J. Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[GM82] J. A. Goguen and J. Meseguer, Security policies and security models. IEEE Symposium on Security and Privacy 1982: 11-20.

[HAR1] R. Harper, F. Honsell, G. Plotkin, *A framework for defining logics*. Journal of the Association for Computing Machinery, 40(1):143-184, January 1993

[HR98] N. Heintze and J. G. Riecke, The slam calculus: programming with secrecy and integrity. POPL 1998: 365-377.

[GK87] G. Kahn, Natural Semantics, STACS 1987:22-39.

[MCGW02] G. Morrisett, K. Crary, N. Glew, and D. Walker, Stack-based typed assembly language. J. Funct. Program. 13(5): 957-959 (2003).

[MED06] R. Medel, *Typed Assembly Language for Software Security*. PhD thesis, Stevens Institute of Technology, 2006. Forthcoming.

[ML98] A. Myers and Barbara Liskov, A Decentralized Model for Information Flow Control. SOSP 1997: 129-142.

[MNZZ99] A. C. Myers, N.Ñystrom, L. Zheng, and S. Zdancewic, JIF: Java information flow, 1999.

[MOR85] G. Morrisett, *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1985. Published as CMU Tech Report CMU-CS-95-226.

[MO01] F. Schneider, G. Morrisett, R. Harper, Language Based approach to Security. Informatics 2001: 86-101.

[MYE99] A. C. Myers, JFlow: Practical Mostly-Static Information Flow Control. POPL 1999: 228-241.

[NEC01] G. Necula Compiling with Proofs, PhD thesis. Carnegie Mellon University, Septiembre 1998.

[NL97] G. Necula and P. Lee, The Design and Implementation of a Certifying Compiler. PLDI 1998: 333-344.

[PIE02] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 2002. Primera Edición.

[PI01] Benjamin C. Pierce, Advanced Topics in Types and Programming Languages. *MIT Press*, 2004. Primera Edición.

[PL81] G. Plotkin, *A Structural Approach to Operational Semantics*. Lecture notes, DAIMI FN-19, Aarhus University, Denmark, 1981.

[POP] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, S. Zdancewic, TALx86: A Realistic Typed Assembly Language" ACM SIGPLAN Workshop on Compiler Support for System Software, 1999: 25-35.

[PS02] F. Pottier and V. Simonet, Information flow inference for ML. ACM Trans. Program. Lang. Syst. 25(1): 117-158 (2003).

[RUS08] A. Russo *Language Support for Controlling Timing-Based Covert Channels*. PhD thesis, Chalmers University of Technology, Sweden 2008.

[SAB01] A. Sabelfeld, The impact of synchronisation on secure information flow in concurrent programs. Ershov Memorial Conference 2001: 225-239.

[SIM03] V. Simonet, Flow caml in a nutshell, 2003. http://cristal.inria.fr/ simonet/soft/flowcaml/

[SM03] A. Sabelfeld and A. C. Myers, Language-based information-flow security. IEEE Journal on Selected Areas in Communications. Vol. 21 (2003).

[SS05] A. Sabelfeld, D. Sands *Dimensions and principles of declassification*. In CSFW 2005: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW 2005), pages 255-269. IEEE Computer Society, 2005

[SV98] G. Smith and D. Volpano, Secure information flow in a multithreaded imperative language. POPL 1998: 355-364.

[UML04] F. Bavera, M. Nordio, R. Medel, J. Aguirre, G. Baum, M. Arroyo, Un Survey sobre Proof-Carrying Code. *5to Simposio Argentino de Computación, AST 2004. Universidad de Córdoba (Argentina)*, 2004.

[VS97] D. Volpano and G. Smith, Eliminating covert flows with minimum typings. CSFW 1997: 156-169.

[VSI96] D. Volpano, G. Smith, and C. Irvine, A sound type system for secure flow analysis. Journal of Computer Security 4(2/3): 167-188 (1996).

[WAL99] J. Morrisett, D. Walker, K. Crary, N. Glew, From system F to typed assembly language. ACM Trans. Program. Lang. Syst. 21(3): 527-568 (1999).

[YI06] D. Yu and N. Islam, A typed assembly language for confidentiality. ESOP 2006: 162-179.

[YU07] D. Yu, More typed assembly language for confidentiality. APLAS 2007: 86-104.