FAMAF Facultad de Matemática, Astronomía, Física y Computación

 UNC Universidad Nacional de Córdoba

Soporte para ARM en un compilador verificado

Trabajo especial de licenciatura

Santiago Arranz Olmos

Diciembre 2022

Dirigido por Miguel Pagano
Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba

Abstract

This work is a study on the Jasmin programming language, used to develop high-speed high-assurance cryptography, as well as a proposal for an extension to add support for new hardware architectures such as ARM Cortex M4. We study the problem of developing critical security software, in particular cryptography, as well as some of the theoretic foundations and tools used to specify and implement these systems. Then, we describe the Jasmin programming language, its compiler and the formal verification of the latter; finally, we report on a generalization proposed by us to adapt the compiler to new cases of interest.

Resumen

Este trabajo es un estudio de un lenguaje de programación, llamado Jasmin, utilizado para desarrollar criptografía eficiente y confiable, así como una propuesta de una extensión a esta herramienta para agregar soporte para nuevas arquitecturas de *hardware* como ARM Cortex M4. Se estudia el problema de desarrollar *software* crítico con aplicaciones en seguridad, en particular criptografía, y se describen brevemente algunos de los fundamentos y herramientas utilizados para especificar e implementar estos sistemas. Luego, se describen el lenguaje de programación Jasmin, su compilador y la verificación formal de este último; y por último una generalización del compilador para adecuarlo a nuevos casos de interés.

Agradecimientos

Acá va uno.
Acá va otro.
Este es el último.

Índice general

1 Introducción

La ciberseguridad es de vital importancia. El término ciberseguridad, o seguridad informática, se utiliza en gran variedad de contextos, pero en general refiere a la protección de bienes tecnológicos e información de daño, robo, o uso no autorizado. Sintéticamente, busca asegurar la confidencialidad, la integridad y la disponibilidad de dichos bienes e información. Mientras que el *hardware* suele protegerse de la misma manera que se resguardan otros bienes materiales, el *software* y la información que este procesa requieren otras medidas [27, 49].

La seguridad informática ha tomado particular relevancia debido al uso generalizado de tecnología digital para soportar el funcionamiento de servicios como hospitales, bancos, fábricas y sistemas de transporte. Por otro lado, la introducción de la computadora personal y del internet hicieron que su alcance se multiplique, pues más y más aspectos de la vida diaria se ven de alguna manera influenciados por estas herramientas. En conclusión, la necesidad de ciberseguridad surge de la dependencia del mundo actual de tecnología digital y de la confidencialidad de la información que esta maneja.

Un componente clave de la ciberseguridad es la criptografía, ya que posibilita la confidencialidad e integridad de información. La criptografía es ubicua en las soluciones tecnológicas modernas, desde páginas web y tarjetas de crédito hasta infraestructura militar.

A continuación se discuten brevemente y a muy alto nivel algunos conceptos fundamentales de la criptografía, intentando dimensionar la importancia y complejidad del área. Luego, en la Sección 1.2, se mencionan algunas aplicaciones de protocolos criptográficos y se examina la tarea de implementar dichos protocolos. Por último, en la Sección 1.3, se describen los temas principales de este trabajo: una propuesta para implementar criptografía de manera fiable y eficiente y una extensión a la misma.

1.1. La criptografía

La criptografía trata sobre la comunicación en presencia de adversarios [48]. Consiste en el diseño y análisis de protocolos resistentes a la influencia de estos adversarios [10].

Mientras que en el pasado la criptografía era un arte, una serie de heurísiticas ad-hoc basadas en lo que se suponía, o esperaba, hacía a la seguridad de un sistema, hoy es una ciencia basada en la matemática y en las ciencias de la computación.

Este cambio puede adjudicarse a Shannon [51], quien asienta las bases de la teoría de la información dando un modelo matemático riguroso de la comunicación de información. Usando estas nuevas herramientas, propone una definición rigurosa de lo

que significa privacidad en [52], conocida como «seguridad perfecta» (*perfect secrecy* en inglés). En cierto sentido, la seguridad perfecta es lo mejor a lo que se puede aspirar: un protocolo criptográfico es perfectamente seguro si un adversario que intercepta un mensaje encriptado no puede deducir nada sobre el contenido del mensaje. Más precisamente, la probabilidad de que la encriptación interceptada provenga de cierto mensaje es igual a la de que provenga de otro mensaje cualquiera. En el mismo trabajo, Shannon probó que el esquema «libreta de un solo uso» (usualmente llamado OTP, por *one time pad* en inglés) es el único sistema perfectamente seguro. Lamentablemente, el OTP no es viable en la práctica, por lo que hoy no se utiliza el paradigma de Shannon. En cambio, se usa una noción distinta de seguridad, que si bien no es perfecta, es suficiente [10].

Al presente la criptografía se define computacionalmente, y resuelve el problema anterior introduciendo una nueva dimensión a la ecuación: el poder de cómputo del adversario. Se busca que un sistema sea seguro ante todo adversario de alguna forma acotado en su poder de cómputo. Así, los sistemas puede romperse *en teoría*, pero no en la práctica. Los ataques son *teóricamente* posibles, pero prácticamente inviables. Las garantías de seguridad en este contexto son al estilo de «en tiempo t, la probabilidad de romper el sistema es menor a $\frac{t}{2^{200}}$» [10]. A continuación se describe brevemente la terminología básica usual en la criptografía moderna, como en las introducciones de Rivest [48], Bellare y Rogaway [10] y Boneh y Shoup [14], y se refiere a dichos trabajos para más detalles.

La comunicación se da entre *partes*, que desean enviarse *mensajes* a través de un *medio* a pesar de la presencia de un *adversario*. La primer tarea de interés es que una parte envíe un mensaje m que pertenece al espacio de mensajes $\mathcal{M}$ a otra. Para ello, se acuerda previamente entre las partes un *secreto compartido* o *clave* k del espacio de claves $\mathcal{K}$, desconocido para el adversario, y también un *protocolo* o *esquema* de encriptación (E, D) tal que $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ y $D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$, donde $\mathcal{C}$ es el espacio de textos cifrados, y $D(k, E(k, m)) = m$ para cualquier clave k y mensaje m. La manera en la que una de las partes, usualmente personificada y llamada Alice, envía m a la otra parte, usualmente llamada Bob, es enviando $c = E(k, m)$, para que Bob recupere el mensaje $m = D(k, c)$. La tarea del adversario es determinar m conociendo todo lo anterior excepto m y k.

Naturalmente, el adversario siempre puede elegir un m arbitrario y lograr su tarea satisfactoriamente con probabilidad $\frac{1}{|\mathcal{M}|}$, y más aún, en otros modelos también puede probar exhaustivamente todas las posibles claves, utilizando tiempo proporcional a $|\mathcal{K}|$. De ahí que lo único que se require del adversario es, similarmente al ejemplo dado antes, que utilice a lo sumo tiempo polinomial en $n = \log_2(|\mathcal{K}|)$ (la cantidad de bits necesaria para representar la clave) y que acierte con probabilidad *no negligible* en n (que no pueda ser acotada por el recíproco de un polinomio en n).

Como es de esperar, las tareas y poderes de las partes y del adversario cambian de acuerdo a lo que se desee modelar. Por ejemplo, se puede permitir que el adversario conozca la encriptación $E(k, m_i)$ de una serie de mensajes $m_1, \ldots, m_q$ de su elección, y luego perdirle que recupere un mensaje distinto a estos $m \notin \{m_1, \ldots, m_q\}$ disponiendo de su encriptación $E(k, m)$. O puede que su tarea no sea recuperar un mensaje m,

sino suplantar a Alice y enviar un mensaje m' a Bob convenciéndolo de que proviene de Alice. Una definición presente en todas las introducciones a la criptografía es *seguridad semántica*, donde el adversario elige dos mensajes m y m' y se los envía a Alice, Alice procede a elegir uno, encriptarlo y responder con la encriptación, para que el adversario intente entonces adivinar cuál de los dos mensajes eligió Alice.

Hay mucha variedad en las propiedades que se busca garantizar con un protocolo. Se mencionaron *privacidad*, en donde el adversario no puede aprender nada de m conociendo $E(k, m)$, y *autenticación*, en donde el adversario no puede suplantar a una de las partes. Otra propiedad importante es *integridad*, donde el adversario es incapaz de modificar el mensaje en tránsito sin ser descubierto. Cada una de estas propiedades implica dar más poder al adversario y requerir más del esquema de encriptación.

Es notable que se pongan tan pocas restricciones sobre el adversario, solo asumiendo que es un algoritmo polinomial, pero en cierto modo esto refleja la realidad. Los ataques a protocolos criptográficos modernos son tan variados como los protocolos mismos, sino más, utilizando nuevos avances en tecnología para disponer de poder de cómputo órdenes de magnitud mayor al estado del arte cuando se diseñó el esquema, errores en las implementaciones criptográficas para lograr el malfuncionamiento de los sistemas, detalles del *hardware* para obtener información acerca de la ejecución de los protocolos, entre muchos otros vectores de ataque. Cada decisión involucrada en el diseño de un protocolo abre la puerta a una clase nueva de ataques, sea al incluir o excluir cierta característica, por lo que es imperativo modelar y analizar el sistema de la manera más abstracta posible.

Asimismo, una base teórica rigurosa es esencial para este área, pues las construcciones criptográficas son «frágiles» (*brittle* en inglés), en el sentido en que son resistentes en estado íntegro, pero una mínima falla hace que todo el sistema se derrumbe. Un sistema puede verse completamente comprometido por la más insignificante vulnerabilidad. Para mantener este nivel de abstracción y rigurosidad, la criptografía se apoya en áreas varias y diversas de la matemática y las ciencias de la computación, como por ejemplo teoría de números, de complejidad, de la probabilidad y de juegos.

1.2. Aplicación e implementación de la criptografía

Prácticamente toda la tecnología actual usa o depende de alguna construcción criptográfica. Toda comunicación privada a través de internet requiere algún tipo de protocolo de encriptación y autenticación, siendo los correos electrónicos, la mensajería instantánea y el comercio electrónico solo algunos ejemplos. Los pagos con tarjeta con chip, las criptomonedas y todo sistema que trate con información confidencial, como bancos, hospitales e instituciones gubernamentales, requieren proteger sus datos contra accesos y alteraciones no autorizadas. Los estados y las empresas aseguran sus comunicaciones e inteligencia encriptándolas. Cada una de estas aplicaciones requiere múltiples construcciones criptográficas para operar satisfactoriamente. Por ejemplo, al menos cinco protocolos están involucrados en una típica conexión Wi-Fi (IEEE 802.11): WPA en la capa de enlace de datos, IPsec en la de red, TLS en la de

transporte y PGP, SSH o S/MIME en la de aplicación.

El extenso uso de la criptografía hace que su diseño e implementación sea crítico. Las vulnerabilidades no solo se explotan para delitos a pequeña escala como robo de dinero, robo de identidad o secuestro de datos, sino también a gran escala: está en el interés de organizaciones o entidades poderosas promover diseños e implementaciones intencionalmente incorrectas de estándares criptográficos. Por ejemplo, en [12] se describe cómo Dual EC, un generador de números pseudoaleatorios, fue estandarizado por NIST (*National Institute of Standards and Technology*, Instituto Nacional de Estándares y Tecnología de los Estados Unidos), ANSI (American National Standards Institute, Instituto Nacional Estadounidense de Estándares) e ISO (*International Organization for Standardization*, Organización Internacional de Normalización) en 2006, a pesar de sus conocidas deficiencias, debido a la influencia de la NSA (*National Security Agency*, Agencia Nacional de Seguridad de los Estados Unidos). Gracias a los documentos filtrados por Edward Snowden en 2013, se descubrió que la agencia no solo diseñó el algoritmo con un *backdoor* para atacar aquellos sistemas que lo utilicen, sino que también pagó diez millones de dólares a RSA Security, una firma de tecnología criptográfica muy importante fundada por tres eminencias de la criptografía, Ron Rivest, Adi Shamir y Leonard Adleman, para que implemente el estándar en su librería BSAFE, una de las más populares del área [6, 45, 36]. La página oficial de NIST [17] informa que al menos 81 productos incluyeron o aún incluyen una implementación de Dual EC validada por la entidad. Más ejemplos de vulnerabilidades y explotaciones de todo tipo pueden encontrarse en [50, 37, 2].

Esta cantidad y diversidad de vulnerabilidades viene de la enorme superficie de ataque que presenta la cadena compuesta por el diseño de protocolos, la implementación de librerías y la implementación de aplicaciones. Como se mencionó antes, la cualidad frágil de la criptografía, y de la seguridad en general, hace que haya razones para enfocarse en cualquiera de estas etapas.

Por ejemplo, en Lazar y col. [37] se analiza una selección de *bugs* relacionados con criptografía reportados en la base de datos CVE [25]. El 17 % de dichos *bugs* se encontraban en código de librerías, mientras que el 83 % restante en código de aplicaciones, pero como el impacto de los errores en protocolos y librerías es mucho mayor que el de los de aplicaciones, se aconseja hacer énfasis en la mitigación de los primeros. Además, se argumenta que los *bugs* en las aplicaciones son prevenibles mediante un mejor diseño de las APIs de las librerías y el uso de herramientas como análisis estáticos.

Sin embargo, es innegable que los errores de implementación invalidan incluso al protocolo mejor diseñado: en el mundo real son las implementaciones las que se ejecutan, no los algoritmos. En Anderson [2] se provee evidencia y sugiere que los errores de implementación son los principales culpables de la mayoría de los ataques. Sostiene que la mayor parte del esfuerzo e investigación en criptografía y seguridad se aplica a cuestiones de marginal relevancia, debido a la preocupación por lo que *puede* salir mal, en lugar de lo que *es probable* que salga mal. Así, se concluye que el acento debe cambiarse a incrementar la fiabilidad de las implementaciones en general. Estos dos trabajos evidencian que el correcto funcionamiento de todos los componentes del

entorno criptográfico son esenciales.

Por último, no hay que perder de vista que el *software* criptográfico es tan solo una parte del contexto tecnológico. El *hardware*, los sistemas operativos, las aplicaciones y los usuarios constituyen una superficie de ataque inmensamente más amplia. Es mucho más sencillo y barato engañar a un usuario mediante *phishing* que romper el esquema criptográfico utilizado por su banco. Es mucho más eficiente espiar y controlar a una población analizando los metadatos de su comunicación, es decir información como con quién, cuándo y desde dónde se comunica, que desencriptar sus mensajes. Es condición necesaria que el *hardware* y el *software* en general sean abiertos y confiables para disminuir estos riesgos.

1.3. La propuesta

La primer parte del presente trabajo es un estudio de una herramienta utilizada para implementar criptografía, llamada Jasmin, introducida por Almeida y col. en [1]. Es parte del set de herramientas Formosa [28], que actualmente cuenta con tres proyectos dedicados a la práctica criptográfica, siendo los otros dos EasyCrypt [8, 9] y LibJade [40]. Jasmin es un lenguaje de programación de bajo nivel, provee asignaciones (de manera restringida), estructuras de control estándar (`if`, `for`, `while`) e instrucciones *assembly*. El desarrollo en Jasmin, entonces, da un control fino sobre el resultado de la compilación, siendo su principal caso de uso la implementación de librerías de alto rendimiento y fiabilidad. El compilador de este lenguaje está desarrollado mayormente en Coq, pero tiene componentes en OCaml y está en gran parte verificado, con distintas estrategias, dejando sin verificar solo el *parsing*, *typing* y *pretty-printing*.

El diseño peculiar de Jasmin resulta de dos requisitos fundamentales impuestos sobre las implementaciones de primitivas criptográficas: confiabilidad y eficiencia.

Confiabilidad. Es de vital importancia conseguir un alto grado de certeza de que una implementación criptográfica respeta su especificación teórica, como atestiguan las secciones anteriores. El desafío que supone garantizar la coherencia con la teoría surge de la dificultad de relacionar especificaciones teóricas abstractas con código concreto de bajo nivel, que suele romper estas abstracciones por razones de eficiencia [37]. Esta tarea a veces se denomina *cerrar la brecha semántica*, y está exacerbada por el hecho de que ambos dominios requiren conocimiento experto especializado, llamado *brecha de competencias técnicas*.

Eficiencia. Es indispensable que las implementaciones criptográficas sean óptimamente eficientes en su utilización de recursos de interés como tiempo de procesamiento, memoria, almacenamiento o uso energético. Primeramente debido a que, como se explica en Brumley y col. [15], estas implementaciones *proveen* seguridad a *otras* aplicaciones; es decir, no proveen funcionalidades utilizadas por usuarios, sino que facilitan componentes para el correcto funcionamiento de otros sistemas, por lo que

desde el punto de vista de la usabilidad son pura sobrecarga. Además, suelen utilizarse en contextos extremos como la encriptación completa de un disco o de todo el tráfico de un servidor VPN. Debido al frecuente uso de dichas implementaciones, la más mínima ineficiencia se multiplica rápidamente y causa diferencias notables. En ese mismo trabajo se menciona que los usuarios suelen desactivar medidas de seguridad si esto mejora la fluidez y usabilidad de las aplicaciones. Es importante resaltar que el mayor riesgo de la criptografía es que no se use [11]. Dicho esto, tampoco se debe comprometer la seguridad por eficiencia. Segundamente, la criptografía se usa en una multitud de dispositivos muy variados, muchos de ellos tienen restricciones estrictas en su capacidad de cómputo, en su cantidad de memoria o su uso energético, como por ejemplo microprocesadores en sistemas embebidos. Para lograr eficiencia y optimizaciones a este nivel es necesario tener conocimiento experto especializado.

Como resultado de los dos puntos anteriores, Jasmin provee al desarrollador con el máximo control de bajo nivel posible sobre el código, así como la máxima expresividad para razonar sobre las implementaciones.

El control de bajo nivel es necesario tanto para la confiabilidad como para la eficiencia de implementaciones criptográficas. Los detalles específicos de la arquitectura para la que se desarrolla afectan directamente la seguridad, ya que determinan cómo un adversario puede, o no, interactuar con la ejecución de una implementación. Por ejemplo, si se sabe que un procesador expone una línea de su caché al ejecutar cierta instrucción, se puede mitigar esta vulnerabilidad evitando dicha instrucción o sobreescribiendo la caché antes de utilizarla, tareas imposibles, de hecho inexpresables, en lenguajes de programación de alto nivel. Por otro lado, anteriormente se justificó por qué es imprescindible desarrollar código óptimamente eficiente, y esto solo puede lograrse con control de bajo nivel. Un compilador nunca podrá equipararse con un experto a la hora de optimizar código de aplicación específica, como el que se usa en criptografía.

Las garantías que se tengan sobre las implementaciones son fundamentales para asegurar su adherencia a las especificaciones teóricas; pero cerrar las brechas semántica y de competencias técnicas no es tarea fácil. Surge entonces la pregunta ¿cómo asegurar la confiabilidad de una implementación criptográfica? Para el caso del algoritmo abstracto se pueden obtener garantías teóricas, pero para la implementación de bajo nivel la situación es más compleja. Las estrategias utilizadas usualmente para incrementar la fiabilidad del *software* no son suficientes, y es inevitable recurrir a las pruebas formales:

- Buenas prácticas y revisión de código. La brecha de competencias técnicas y la dificultad inherente al dominio hacen que esta estrategia no brinde garantías suficientes. Además, aplicar este tipo de recomendaciones suele ser difícil en sistemas complejos, con requerimientos frecuentemente en conflicto, que deben interoperar con sistemas externos, como es el caso de las implementaciones criptográficas [37].

- *Testing* tradicional y *fuzzing*. El *testing* tiene una limitación ineludible: solo puede demostrar la existencia de errores, no su ausencia. Aún así, en ciertos

casos se puede llegar a establecer una confianza razonable mediante este método, sobre todo cuando se lo acompaña con *fuzzing* y análisis estáticos, pero aquí entran en juego otras limitaciones. Primero, no es fácil identificar o caracterizar qué es un *bug* en el contexto de la seguridad. Las vulnerabilidades vienen en todo tipo de formas, y suelen involucrar violaciones de un invariante complejo, más que un *crash* del *software* [37]. Segundo, está la dificultad inevitable de encontrar errores que se evidencien poco frecuentemente. Los casos de *test* usualmente involucran escenarios ad-hoc que previenen errores ya cometidos o conocidos [37]. En el caso de uso de la seguridad criptográfica, todas las vulnerabilidades deben descartarse, frecuentes e infrecuentes, previstas e imprevistas.

- Análisis estáticos: detección de anti-patrones, ejecución simbólica, e interpretación abstracta. Estas estrategias pueden descartar clases de vulnerabilidades de manera confiable, e incluso asegurar corrección funcional en algunos casos, pero la gran variedad en los contextos, implementaciones, y propiedades del área de la criptografía hacen que virtualmente sea necesario implementar una herramienta nueva para cada par especificación-implementación. Por esto, si bien Jasmin se sirve de ciertos análisis estáticos, debe proveer otras herramientas para dar garantías suficientes sobre el código.

- Pruebas a mano: en papel y chequeadas por computadora. Estas estrategias proveen la mayor generalidad, flexibilidad, y fiabilidad al precio de un volumen mucho mayor de trabajo manual. Acotando el espacio de casos de uso a las pruebas criptográficas usuales, se puede obtener una herramienta que balancea razonablemente ambos aspectos. Además, está la ventaja de que en criptografía las especificaciones de los algoritmos se expresan de manera rigurosa desde un principio. Si bien las pruebas en papel son técnicamente suficientes, chequearlas es inviablemente tedioso y propenso a errores. Una herramienta para escribir pruebas y chequearlas automáticamente resuelve este problema, y además simplifica varias otras tareas mecánicas involucradas. Existen varias herramientas que automatizan considerables partes de esta labor, por ejemplo mediante el uso de SMTs. Sin embargo, la automatización tiene un límite, pues probar propiedades para *inputs* de tamaño arbitrario requiere razonamientos abstractos e inductivos. Además, la escala de los sistemas en cuestión hacen que sea un requisito probar propiedades modularmente para controlar la complejidad de la tarea [37].

La formalización de estas implementaciones no solo contribuye a su confiabilidad, sino que también puede informar el diseño de las APIs que ofrecen a las aplicaciones, minimizando el riesgo de su uso erróneo [37]. Más aún, también hace a la eficiencia, ya que sino las protecciones que se suelen utilizar para garantizar seguridad son incompletas, o innecesariamente conservadoras [16].

El diseño de Jasmin intenta balancear el control de bajo nivel y la viabilidad de razonamiento abstracto. Se busca proveer un lenguaje de bajo nivel, tan cercano a *assembly* como sea posible, restringiendo aquellas características que impiden el

razonamiento de alto nivel sobre el programa. Como resultado, Jasmin genera no sólo código *assembly* sino también definiciones para modelar la implementación en EasyCrypt. En el Capítulo 3 se estudia Jasmin en más detalle.

Actualmente el compilador de Jasmin solo produce código *assembly* para la arquitectura x86-64, esto significa que puede desarrollarse *software* para la mayoría de las computadoras personales y supercomputadoras. Sin embargo, se espera que en el futuro otras arquitecturas, como ARM o RISC-V, tomen prevalencia, ya que son utilizadas en sistemas embebidos [54, 42] por su bajo coste y consumo de recursos, en dispositivos móviles, área en la que ARM lidera por un amplio margen con más del 90 % del mercado en 2020 [34], y prometen ser útiles también para nuevas supercomputadoras [24, 46]. A continuación se describen las principales propiedades de uno de los procesadores de la familia ARM.

El procesador ARM Cortex-M4 es un procesador de bajo consumo energético, baja cantidad de puertas lógicas, y baja latencia de interrupciones. Está diseñado para sistemas embebidos que requieren rápida respuesta a interrupciones. El procesador implementa un *pipeline* de tres etapas, especulación de saltos, y el set de instrucciones Armv7E-M [3]. El set de instrucciones es de tipo RISC (*reduced instruction set computer*, computadora con set de instrucciones reducido) y está apuntado a microcontroladores para los que el tamaño y el determinismo de las implementaciones son más importantes que la eficiencia general [4]. Los registros y las palabras en memoria son de 32 bits, y la interacción con la memoria se hace únicamente con instrucciones *load* y *store*, si bien existen instrucciones de este tipo que procesan varias palabras al mismo tiempo.

Teniendo en cuenta las posibles aplicaciones del soporte para ARM Cortex-M4 en Jasmin, y de otras arquitecturas en general, en la segunda parte de este trabajo se propone una generalización del compilador de Jasmin para soportar otras arquitecturas y una extensión para agregar soporte para esta arquitectura.

El desarrollo descripto en este trabajo se realizó en el marco de una pasantía en el Instituto Max Planck de Seguridad y Privacidad [41] entre junio 2021 y agosto 2022, bajo la supervisión de Peter Schwabe y Gilles Barthe.

2 Las herramientas

El compilador de Jasmin, que será el objeto principal de los siguientes capítulos de este trabajo, está desarrollado en Coq, un sistema de manejo de pruebas formales. A continuación se discuten brevemente los fundamentos teóricos en los que se basan las herramientas que se introducirán más adelante.

Cuando se dice que el compilador de Jasmin está verificado se está afirmando que su comportamiento esperado se encuentra de alguna manera especificado, y que se cuenta con una demostración de que la implementación del compilador se corresponde con dicha especificación. Por ejemplo, si el compilador debe fallar al compilar la instrucción `x := y / 0`, parte de la especificación será algo como «`compile (x := y / 0)` debe ser `Fail` para cualesquiera `x` e `y` variables», y luego se debe mostrar que estos casos se detectan de alguna forma y que el compilador falla al hacerlo.

Para lograr verificar un programa, se debe de alguna manera poder expresar su especificación, su implementación, y la prueba que las pone en correspondencia. En la sección que sigue se introduce una manera de relacionar proposiciones, programas y pruebas.

2.1. Proposiciones como tipos

Proposiciones como tipos (*Propositions as Types* en inglés), a veces llamado el isomorfismo de Curry-Howard y de varias otras maneras, es una de las ideas más importantes de la computación y la lógica intuicionista. Relaciona de manera íntima estas dos áreas, específicamente la teoría de tipos y la teoría de pruebas, e incluso puede extenderse e incluir también la teoría de categorías. Esencialmente, consiste en identificar la relación entre un tipo y un elemento de ese tipo con aquella entre una proposición y una prueba de dicha proposición. Los tipos actuan como «clasificadores» de términos, en el sentido en que un tipo divide a los términos en tipables o no tipables por él, basándose en la estructura de estos; y las proposiciones actúan como clasificadores de pruebas de manera análoga.

En 1934 el matemático alemán Gerhard Gentzen propuso la *deducción natural* como un cálculo de pruebas para realizar razonamientos lógicos de manera «natural» [31]; esto fue en respuesta a las axiomatizaciones de la matemática del momento, que eran poco intuitivas y lejanas a lo que una persona hace al razonar lógicamente. Este cálculo se presenta mediante reglas de deducción, relacionando fórmulas formadas por las constantes y conectivos lógicos usuales, conjunción ($A \wedge B$), disyunción ($A \vee B$), implicación ($A \Rightarrow B$), y falsedad ($\bot$). La deducción natural de Gentzen también incluye la regla de reducción al absurdo y los cuantificadores $\forall$ y $\exists$, pero no serán considerados en esta sección.

Gentzen observó que su presentación natural de la lógica consiste en reglas de *introducción* y de *eliminación* para cada conectivo, correspondiendo las primeras a una «definición» del conectivo y las últimas a las consecuencias de dicha definición[1]

> The introductions represent, as it were, the "definitions" of the symbols concerned, and the eliminations are no more, in the final analysis, than the consequences of these definitions. (Gentzen [31, pág. 295])

Esta observación es clave para la demostración de consistencia de la deducción natural, que se basa en probar la propiedad que Gentzen llama *Hauptsatz*, que dice, intuitivamente, que toda deducción puede reducirse a una forma *normal*, en la que «solamente ocurren en la prueba conceptos contenidos en la conclusión»

> The Hauptsatz says that every purely logical proof can be reduced to a determinate, though not unique, normal form. Perhaps we may express the essential properties of such a normal proof by saying "it is not roundabout". No concepts enter into the proof other than those contained in its final result, and their use was therefore essential to the achievement of that result. (Gentzen [31, pág. 289])

Un corolario de esto es que una prueba de $\bot$ sin hipótesis es imposible, ya que al normalizarla se obtendría una en la que solo ocurren sus subfórmulas, pero $\bot$ no tiene subfórmulas.

Casi simultáneamente, en 1932, el matemático estadounidense Alonzo Church formuló el *cálculo lambda* para formalizar las fundaciones de la lógica [19]. Su sintaxis puede verse en la Figura 2.1 a la izquierda. Sobre los términos del cálculo lambda se suelen definir distintas relaciones como la *equivalencia alfa*, la *reducción beta*, y la *contracción eta*. La más importante es la reducción beta, que se define como sigue:

$$(\lambda x.\, M)\, N \Rightarrow_\beta M[x := N]$$

donde M y N son términos del cálculo lambda, y $M[x := N]$ es el término que resulta de substituir las ocurrencias libres de x por N en M. Esta regla utiliza todas las componentes del cálculo lambda: variables, aplicación y abstracción, y expresa

[1] Merece la pena notar que la observación sobre las reglas de introducción y eliminación puede hacerse de manera dual, entendiendo a las últimas como definiciones del conectivo y a las primeras como consecuencias. En Dummet [26, pág. 287] se refiere a estos dos puntos de vista como las miradas *verificacionista* y *pragmatista* del significado de los conectivos lógicos. La mirada verificacionista sostiene que el significado de una sentencia lógica proviene de lo que se necesita para afirmarla (la introducción del conectivo), y que las consecuencias que se pueden hacer con esta sentencia como premisa (la eliminación) quedan así determinadas (es decir, no se puede extraer más información de la sentencia que la que se usó para introducirla). Por otro lado, la mirada pragmatista consiste en entender al significado de una sentencia como dado por las consecuencias a las que esta permite llegar, y que esto determina entonces lo necesario para concluir dicha sentencia (es decir, no se puede introducir la sentencia con menos información de la que es necesaria para concluir sus consecuencias). Lo anterior es en el contexto de sistemas lógicos cuyas reglas están en armonía, que para los fines de esta aclaración pueden entenderse como cualquier sistema razonable.

```
<expr> ::= <var>                    <expr> ::= <var><type>
         | <expr> <expr>                      | <expr> <expr>
         | λ<var>.<expr>                      | λ<var><type>.<expr>

                                     <type> ::= <typevar>
                                              | <type> → <type>
```

Figura 2.1: Sintaxis del cálculo lambda y del cálculo lambda simplemente tipado.

exactamente la noción de substitución. Así, puede decirse que el cálculo lambda con esta regla se basa esencialmente en la noción de mapeo, en el sentido en que $\lambda x.\, M$ asocia N con $M[x := N]$. Puede decirse entonces que la amplia aplicabilidad del cálculo lambda se debe a la identificación de esta noción con la de función (en ciencias de la computación) y la de consecuencia o implicación (en lógica).

En la medida en que el cálculo lambda es una teoría de funciones, tiene la característica de que estas se definen de manera *intensional*, es decir como un proceso para transformar un argumento en un resultado (esta noción también se llama «funciones como reglas»); contrastando con el concepto usual de función matemática como un conjunto de pares, que se denomina *extensional* [7].

Al poco tiempo de la propuesta de Church, en 1935, dos de sus estudiantes, Stephen Kleene y John Rosser, mostraron que el sistema lógico era inconsistente mediante la paradoja de Kleene-Rosser [35]. Es por esto que, en 1940, Church introdujo el *cálculo lambda simplemente tipado* [18]; su sintaxis se muestra en la Figura 2.1 a la derecha. La sintaxis, y el formalismo, puede extenderse agregando las construcciones de tipos usuales, como el producto de dos tipos `<type> * <type>`, la unión disjunta `<type> + <type>` y el tipo vacío `Empty`.

En la Figura 2.2 se muestran las reglas de inferencia de la deducción natural propuestas por Gentzen y las reglas de tipado de la extensión del cálculo lambda simplemente tipado de Church. Las reglas de inferencia estan etiquetadas como se mencionó anteriormente, como reglas de introducción (por ejemplo $I_\wedge$ o $I^l_\vee$) o eliminación (por ejemplo $E^l_\wedge$ o $E_\vee$). El *contexto* Γ en el caso del cálculo lambda es un *contexto de tipado* con ligamientos de variables y sus tipos, como $\{x : A, y : A \to A, \ldots\}$.

El propósito de la comparación en la Figura 2.2 no es señalar una mera coincidencia. Por un lado, ambos sistemas son sistemas de lógica formal, por lo que no es sorprendente que tengan similaridades. Cabe destacar que se omitieron ciertas partes de la deducción natural (reducción al absurdo y cuantificadores) que no se corresponden con este cálculo lambda pero sí con otras variantes. Y por otro lado, existen más ejemplos de correspondencias entre sistemas originados en la lógica formal y la teoría de tipos, como Girard–Reynolds [56].

Lo que se busca ilustrar es, en cambio, que estas correspondencias son consecuencias de un concepto más abstracto: proposiciones como tipos. Proposiciones como tipos es, como se mencionó anteriormente, la identificación de la relación entre un tipo y un elemento de ese tipo con aquella entre una proposición y una prueba de dicha

proposición. De la misma manera en que una proposición puede interpretarse como el conjunto de todas sus pruebas, un tipo puede interpretarse como el conjunto de todos los términos de ese tipo.

Hay formalismos lógicos que están diseñados para explotar esta relación de proposiciones como tipos; el que se describe a continuación, Coq, es uno muy conocido. La implementación de *software* en Coq permite entrelazar el desarrollo del código con su especificación e incluso su verificación. Si bien esto añade ciertas dificultades o inconveniencias, especificar, probar e implementar un sistema en un mismo lenguaje da lugar a una uniformidad, modularidad y reuso del código muy agradables.

2.2. Coq

Coq [53] es un sistema de manejo de pruebas formales con tres componentes principales: un lenguaje de especificación para modelar, especificar e implementar sistemas; un asistente de pruebas, para construir y chequear pruebas formales; y un extractor de programas, que sintetiza programas basándose en una especificación.

El lenguaje de especificación de Coq, llamado Gallina, está basado en el formalismo lógico CIC [23] (*Calculus of Inductive Constructions*, Cálculo de Construcciones Inductivas), y su principal propósito es facilitar la construcción de términos CIC. Así, tanto la implementación como los tipos de datos, proposiciones y pruebas involucradas en un desarrolo en Coq son simplemente términos CIC.

> **CIC.** En 1988 Thierry Coquand presentó un nuevo formalismo constructivo, el Cálculo de Construcciones (CoC) [21, 22], que es una generalización del cálculo lambda simplemente tipado [18] en tres aspectos
>
> - Polimorfismo paramétrico. Un término puede depender de un tipo, por ejemplo $\lambda x : A.\, x$ es un término de tipo $A \to A$ para cada tipo A.
> - Operadores de tipos. Un tipo puede depender de un tipo, por ejemplo $A \to A$ es un tipo para cada tipo A.
> - Tipos dependientes. Un tipo puede depender de un término, por ejemplo $\texttt{Vec nat}\ n$ es el tipo de los vectores de naturales con n elementos para cada n.
>
> CoC es un formalismo muy expresivo. El listado anterior puede ejemplificarse también mostrando que pueden escribirse funciones que toman términos y devuelven términos (como $\lambda x.\, x + 1 : \texttt{nat} \to \texttt{nat}$), que toman tipos y devuelven términos (como $\lambda A.\, \texttt{nil} : (A : \texttt{Type}) \to \texttt{List}\ A$), que toman tipos y devuelven tipos (como $\lambda A.\, A \to A : \texttt{Type} \to \texttt{Type}$) y que toman términos y devuelven tipos (como $\lambda x.\, \texttt{Vec nat}\ x : \texttt{nat} \to \texttt{Type}$).
>
> Un problema de CoC es que las definciones inductivas pueden resultar inconvenientes ya que suelen involucrar términos bastante complejos (se usa la codificación de Church con polimorfismo para representarlos) [32].

$$\frac{A \qquad B}{A \wedge B}\, I_\wedge \qquad \frac{A \wedge B}{A}\, E^l_\wedge \qquad \frac{A \wedge B}{B}\, E^r_\wedge$$

$$\frac{\Gamma \vdash x : A \qquad \Gamma \vdash y : B}{\Gamma \vdash (x, y) : A * B} \qquad \frac{\Gamma \vdash p : A * B}{\Gamma \vdash \texttt{fst}\ p : A} \qquad \frac{\Gamma \vdash p : A * B}{\Gamma \vdash \texttt{snd}\ p : B}$$

$$\frac{\begin{array}{c}[A]_x \\ \vdots \\ B\end{array}}{A \Rightarrow B}\, I^x_\Rightarrow \qquad \frac{A \Rightarrow B \qquad A}{B}\, E_\Rightarrow$$

$$\frac{\Gamma, x : A \vdash y : B}{\Gamma \vdash \lambda x.\, y : A \to B} \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f\ x : B}$$

$$\frac{A}{A \vee B}\, I^l_\vee \qquad \frac{B}{A \vee B}\, I^r_\vee \qquad \frac{\bot}{A}\, E_\bot$$

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \texttt{inl}\ x : A + B} \qquad \frac{\Gamma \vdash y : B}{\Gamma \vdash \texttt{inr}\ y : A + B} \qquad \frac{\Gamma \vdash x : \texttt{Empty}}{\Gamma \vdash y : A}$$

$$\frac{A \vee B \qquad \begin{array}{c}[A]_x \\ \vdots \\ C\end{array} \qquad \begin{array}{c}[B]_y \\ \vdots \\ C\end{array}}{C}\, E^{x,y}_\vee$$

$$\frac{\Gamma \vdash p : A + B \qquad \Gamma, x : A \vdash z_x : C \qquad \Gamma, y : B \vdash z_y : C}{\Gamma \vdash \begin{array}{l}\texttt{match}\ p\ \texttt{with} \\ \quad \texttt{|}\ \texttt{inl}\ x\ \texttt{=>}\ z_x \\ \quad \texttt{|}\ \texttt{inr}\ y\ \texttt{=>}\ z_y \\ \texttt{end}\end{array} : C}$$

Figura 2.2: Reglas de inferencia de la deducción natural junto con las reglas de tipado del cálculo lambda simplemente tipado.

```
Inductive bool : Set :=
  | true  : bool
  | false : bool.

Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

```
Definition is_O (n : nat) : bool :=
  match n with
  | O   => true
  | S _ => false
  end.

Fixpoint add (n m : nat) : nat :=
  match n with
  | O    => m
  | S n' => S (add n' m)
  end.
```

Figura 2.3: Definición de dos tipos inductivos y dos funciones, una recursiva (`add`) y una no recursiva (`is_O`).

> Debido a esto, Coquand y Paulin-Mohring extendieron CoC con primitivas para definiciones inductivas, dando lugar al Cálculo de Construcciones Inductivas, CIC [23].
>
> Usando proposiciones como tipos se logra tratar de manera uniforme las aplicaciones lógicas y computacionales de CIC, y es usual utilizar las palabras tipo y proposición y término y prueba de manera intercambiable.

Gallina es un lenguaje de programación funcional con las construcciones usuales, en la Figura 2.3 pueden verse ejemplos de definiciones de tipos inductivos y funciones. El comando `Inductive` introduce un tipo inductivo, `Definition` una definición y `Fixpoint` una definición recursiva. Estos comandos forman parte de un lenguaje llamado Vernacular, que se utiliza para interactuar con Coq; por ejemplo para introducir definiciones (con `Inductive`, `Definition`, etc.), imprimir el tipo de un término con `Check`, imprimir un término con `Print` y entrar en modo de prueba con `Proof`.

El asistente de pruebas de Coq permite construir pruebas interactivamente. Las propiedades se enuncian con un comando Vernacular como `Theorem`, `Lemma` o `Remark`, y una vez enunciada la propiedad se entra en *modo de prueba*, donde se tiene un *objetivo* (lo que se desea probar) y un *contexto* (las hipótesis). Las pruebas en Coq se realizan transformando el objetivo mediante *tácticas* hasta que se llega a algo trivial.

En la Figura 2.4 pueden verse ejemplos de teoremas y sus pruebas. La táctica `intro` aplicada a un contexto de la forma `forall (x : A), P` introduce una variable de tipo `A` al contexto y transforma el objetivo en `P`, como puede verse en los comentarios de la figura; esto se corresponde con la introducción del cuantificador universal en deducción natural. Similarmente, la táctica `induction` hace inducción, dando lugar a dos objetivos (caso base y caso inductivo), y agrega las hipótesis apropiadas al contexto (la hipótesis inductiva en el caso inductivo). La táctica `simpl` simplifica el objetivo (usando las definiciones disponibles, la de `add` en el ejemplo); mientras que `rewrite` aplicada a una hipótesis de la forma `x = y` reemplaza en el objetivo todas las ocurrencias de `x` por `y`. Finalmente, la táctica `reflexivity` termina la prueba si el

objetivo es de la forma `x = x`.

Los ejemplos de la Figura 2.4 muestran cómo el lenguaje de tácticas en Coq se utiliza para probar proposiciones, pero teniendo en cuenta proposiciones como tipos, tanto la proposición a probar como la prueba son términos CIC (un tipo y un elemento de ese tipo, respectivamente). Las tácticas sirven para construir términos CIC de manera más sencilla cuando se trata de pruebas, y de hecho, estos términos podrían darse de manera explícita con `Definition` o similares. A continuación se reconsideran los dos ejemplos anteriores con esto en mente.

En el contexto de CIC, se tiene que una prueba de una proposición de la forma `forall (x : A), P` es una función que transforma cada elemento `a` de tipo `A` en una prueba de `P`$[x := a]$. Es por esto que para construir el término `add_O_n` se debe dar una función que transforme a cada natural `n` en una prueba de `add O n = n`. La igualdad proposicional x `=` y solo acepta un tipo de prueba, `eq_refl`, que para cada x de tipo A prueba x `=` x. Su tipo es entonces

```
forall (A : Type) (x : A), x = x
```

y se sigue que el término

```
fun (n : nat) => eq_refl nat n
```
(2.1)

es una prueba de

```
forall (n : nat), n = n
```

es decir, tiene este tipo.

Lo interesante es que al imprimir `add_O_n` con el comando `Print` se obteniene el término 2.1, a pesar de que sus tipos no coincidan a primera vista. Sin embargo, se puede comprobar que estos dos tipos son el mismo: por la definición de `add` en la Figura 2.3 se tiene que

```
n ≡ add O n
```

y entonces

```
forall (n : nat), n = n ≡ forall (n : nat), add O n = n
```

donde ≡ significa *igualdad definicional*, o *igualdad módulo computación*. La igualdad definicional es un concepto muy importante para la teoría de tipos, pero no es necesario para el resto de este trabajo; es suficiente tener presente que para Coq los dos tipos anteriores son equivalentes pues simplemente se debe expandir la definición de `add`. Cuando Coq examina un término en este contexto, lo hace reduciéndolo a su forma normal, teniendo en cuenta todas las definiciones hechas anteriormente.

Por otro lado, como se ve en la Figura 2.5, imprimir `add_n_O` da algo bastante más complejo. Al ser una cuantificación universal, el término también es una función, pero como esta prueba requiere razonar por inducción, se debe recurrir al principio de inducción en los naturales, llamado `nat_ind`. Al definir un tipo con `Inductive`, Coq genera un principio de inducción para ese tipo automáticamente, en este caso `nat_ind` con tipo

```
Theorem add_0_n : forall (n : nat), add 0 n = n.
Proof.
  (* ⊢ forall (n : nat), add 0 n = n *)
  intro n.

  (* n : nat ⊢ add 0 n = n *)
  simpl.

  (* n : nat ⊢ n = n *)
  reflexivity.
Qed.

Theorem add_n_0 : forall (n : nat), add n 0 = n.
Proof.
  (* ⊢ forall (n : nat), add n 0 = n *)
  induction n as [|n' HI].

  (* Caso base. *)
  - (* ⊢ add 0 0 = 0 *)
    simpl.

    (* ⊢ 0 = 0 *)
    reflexivity.

  (* Caso inductivo. *)
  - (* n' : nat, HI : add n' 0 = n' ⊢ add (S n') 0 = S n' *)
    simpl.

    (* n' : nat, HI : add n' 0 = n' ⊢ S (add n' 0) = S n' *)
    rewrite HI.

    (* n' : nat, HI : add n' 0 = n' ⊢ S n' = S n' *)
    reflexivity.
Qed.
```

Figura 2.4: Enunciado y prueba de que el cero es neutro a izquierda y a derecha.

```
1  fun (n : nat) =>
2      nat_ind
3          (fun (m : nat) => add m O = m)
4          (eq_refl nat O)
5          (fun (n' : nat) (HI : add n' O = n') =>
6              eq_ind_r
7                  nat
8                  n'
9                  (fun (m : nat) => (S m = S n'))
10                 (eq_refl nat (S n'))
11                 (add n' O)
12                 HI).
```

Figura 2.5: El término `add_n_O` construido anteriormente.

```
forall (P : nat -> Prop),
  P O
  -> (forall (n : nat), P n -> P (S n))
  -> (forall (n : nat), P n)
```

que se corresponde al principio de inducción usual, con un caso base y uno inductivo, y es utilizado al aplicar la táctica `induction`.

Se ve entonces que en la Figura 2.5 el primer argumento de `nat_ind` (en la Línea 3) es la proposición a probar, y el segundo (en la Línea 4) es la prueba para el caso base, `add O O = O`, que como se mencionó antes es una aplicación de `eq_refl`.

El tercer argumento (en la Línea 5) es más interesante, pues es la prueba del caso inductivo. Por lo discutido anteriormente, debe ser una función que tome un natural `n'` y una prueba de `add n' O = n'` (la hipótesis inductiva) y devuelva una prueba de `add (S n') O = S n'`. Por definición de `add` se tiene que

$$\texttt{add (S n') O} \equiv \texttt{S (add n' O)}$$

y por ende

$$\texttt{add (S n') O = S n'} \equiv \texttt{S (add n' O) = S n'}$$

así que puede utilizarse la hipótesis inductiva `IH` para reescribir el subtérmino `add n' O` por `n'` (en la Figura 2.4 se utiliza la táctica `rewrite` para esto). La reescritura de `x` por `y` dado `x = y` se realiza con el principio de inducción de la igualdad, `eq_ind_r`, sobre el que no se ahondará en esta introducción.

Así es como tanto programas, como tipos, proposiciones, y pruebas en Coq son simplemente términos CIC. Las tácticas son solamente una manera de facilitar la construcción de dichos términos, tarea que en otros asistentes de prueba, por ejemplo Agda [55], debe hacerse explícitamente. Esto da lugar a una simpleza y regularidad muy convenientes en un sistema como Coq, permitiendo implementar y verificar *software* de manera uniforme.

El último componente de Coq es el extractor de programas. Para extraer código de un desarrollo en Coq a un lenguaje de programación convencional como OCaml o Haskell se debe tener en cuenta que estos no son tan expresivos como CIC; deben descartarse ciertas porciones del código Coq, adaptándose acordemente lo restante. En particular, para estos casos la tarea principal es remover los tipos dependientes, ya que tanto OCaml como Haskell tienen polimorfismo paramétrico y operadores de tipos.

El formalismo F_ω de Girard [33] es un sistema con las características deseadas: puede entenderse como CIC sin tipos dependientes. Es por esto que inicialmente la extracción se estudió como el problema de convertir términos CIC en términos F_ω (más precisamente, términos CIC tipables en F_ω). Esta primer versión del mecanismo de extracción fue propuesta por Christine Paulin-Mohring en [44]. Tiene limitaciones importantes, pues requiere introducir dos constantes, `Spec` y `Prop`, denominadas *sorts*, que actúan como tipos de los tipos usuales. Por ejemplo, se tiene que `nat` : `Spec`, `bool` : `Spec`, `True` : `Prop`, y asumiendo x : `nat` se tiene x = x : `Prop`. Estas *sorts* clasifican, respectivamente, a los tipos de los términos *informativos* (que se desean extraer) y a los de los *irrelevantes* (cuya interpretación es meramente lógica). Esta clasificación debe hacerse manualmente, y con cuidado, ya que impactará en el resultado de la extracción. Actualmente el mecanismo de extracción extiende la propuesta de Paulin-Mohring, como puede verse en [38, 39].

3 Jasmin

En este capítulo se describe el lenguaje Jasmin y su compilador. Si bien no se da una sintaxis completa del lenguaje, se estudia una simplificación en la Subsección 3.2.1. Como se mencionó anteriormente, el lenguaje es muy simple, consistiendo en asignaciones, operaciones *assembly* y cuatro estructuras de control estándar: `if`, `for`, `while` y llamadas a función. Se describen la motivación detrás del diseño de este lenguaje y algunas características de interés, así como la estructura del compilador, algunas de sus fases, sus lenguajes internos y su verificación.

3.1. El lenguaje

Jasmin es un lenguaje concebido para implementar criptografía de alto desempeño y confiabilidad. La motivación subyaciente el diseño del lenguaje es desarrollar software de muy bajo nivel (*assembly*), para que sea de alto desempeño, y que sea fácilmente verificable (en EasyCrypt), para dar buenas garantías sobre su comportamiento. Estos dos objetivos entran en conflicto frecuentemente, pues el código de bajo nivel hace uso profuso de efectos secundarios (por ejemplo modificando *flags* del procesador) y de flujo de control no estructurado (por ejemplo `GOTO`s). Por un lado, verificar *software* con efectos secundarios es engorroso porque se debe trabajar con, y mantener actualizado, un estado global, que puede llegar a contener demasiada información para resultar comprensible. Para razonar sobre *software* es ampliamente preferible utilizar construcciones sin efectos secundarios, denominadas funcionales, ya que permiten razonamientos locales: una porción de código sin efectos secundarios hace lo mismo sin importar en qué contexto se use. Y por otro lado, el flujo de control no estructurado descarta la posibilidad de razonar de manera modular y composicional, lo que significa que incluso para código de tamaño moderado la verificación se hace inviable.

La motivación anterior da lugar a una serie de objetivos concretos que apuntan a obtener lo mejor de los dos mundos. El lenguaje se diseña para tener:

- Máximo control sobre el *assembly* generado. De esta manera, se provee la expresividad y flexibilidad necesarias para implementar *software* de alto desempeño. En un programa Jasmin es posible usar directamente instrucciones de la arquitectura para la que se esté programando, siempre que no alteren el flujo de control.
- Flujo de control de alto nivel. Se hace posible entonces la verificación de desarrollos sustanciosos en Jasmin. Junto con el ítem siguiente, esta propiedad permite la verificación *composicional* del código. Este objetivo no debe implicar una pérdida de eficiencia, o al menos no una significativa.

```
param int N = 10;

fn dotp(reg ptr u64[N] v1 v2) -> reg u64 {
  reg u64 res;
  reg u64 tmp;
  inline int i;

  res = 0;
  for i = 0 to N {
    tmp = v1[i];
    tmp *= v2[i];
    res += tmp;
  }

  return res;
}
```

```
movq  $0, %rax
movq  (%rcx), %rsi
imulq (%rdx), %rsi
addq  %rsi, %rax
movq  8(%rcx), %rsi
imulq 8(%rdx), %rsi
addq  %rsi, %rax
...
movq  72(%rcx), %rsi
imulq 72(%rdx), %rsi
addq  %rsi, %rax
ret
```

Figura 3.1: Rutina para calcular el producto punto en Jasmin y el *assembly* generado.

- Efectos secundarios explícitos. Así, el código se vuelve funcional, si bien más verboso.
- Determinismo. Pues hace al razonamiento sobre programas Jasmin es considerablemente más simple.
- Preservación de ciertas propiedades criptográficas. Lo que permite que ciertos análisis, actualmente *memory safety* y *constant-time security*, puedan realizarse sobre el código Jasmin en lugar de sobre el *assembly*, que sería mucho más complicado.

El primero de estos puntos imposibilita tener un lenguaje portable a distintas arquitecturas, al menos para cualquier desarrollo realista. Esta desventaja no es grave, pues las implementaciones optimizadas de rutinas criptográficas no son portables por naturaleza. El segundo, por su parte, impide escribir código que por ejemplo aproveche detalles de la implementación de los `GOTO`s de la arquitectura, o evite un chequeo extra de las *flags*. Si bien esto es ciertamente una limitación, el impacto suele ser reducido.

Como puede verse en la Figura 3.1, Jasmin es un lenguaje de muy bajo nivel. Existe un solo tipo de datos: la palabra de bits; y al programar se debe explicitar el tamaño de las palabras (por ejemplo `U8`, `U16` o `U32`) y su almacenamiento (en un registro o en la pila). Hay construcciones que facilitan el uso del lenguaje como parámetros (`param` en la figura), enteros, booleanos y arreglos, pero todo esto se resuelve en tiempo de compilación. Por ejemplo, los parámetros siempre tienen un valor concreto que se reemplaza antes de compilar, a veces se los llama «constantes con nombre». Si bien los arreglos son una estructura de datos muy importante para el desarrollo de *software* criptográfico, no se vieron involucrados en los cambios presentados en este trabajo, así que se omiten por simplicidad. Los bucles `for` siempre deben tener un contador entero, y son desenrollados por el compilador; en la Figura se ve cómo las

tres instrucciones del cuerpo del bucle (`movq`, `imulq`, y `addq`) se repiten para cada uso de `i` (0, 8, y así hasta 72).

Las funciones en Jasmin puede ser `export`, `inline` o subrutinas. Las primeras son las que interactúan con librerías y código externo, por lo que deben respetar ciertas convenciones, por ejemplo la convención de llamadas de C. Las segundas se eliminan durante la compilación, reemplazando todas sus llamadas por su cuerpo, y las últimas son funciones auxiliares que deben preservarse pero pueden no respetar las convenciones de las `export`.

En otros lenguaje de programación el cuerpo del ciclo podría escribirse

```
res += v1[i] * v2[i]
```

y el compilador o intérprete se encargaría de transformar esta instrucción en las tres de la figura usando un valor intermedio. En Jasmin esto no es posible, puesto que para proveer control absoluto sobre el código generado cada línea de Jasmin se corresponde con exactamente una instrucción *assembly*.

Por otro lado, Jasmin tiene varias ventajas sobre *assembly*. Primero, como puede verse en la Figura 3.2, su sintaxis es similar a la de muchos lenguajes de programación establecidos, por lo que es mucho más clara. Segundo, si bien se provee control absoluto sobre las instrucciones *assembly* de datos, es imposible acceder a las instrucciones de control, lo que significa que el código generado necesariamente tiene un flujo de control estructurado, dado por el compilador. Además, el compilador infiere muchos detalles que, si bien son necesarios para escribir *assembly* válido, no son de interés para desarrollar el código. Por ejemplo, las instrucciones *assembly* deben explicitar el tamaño de sus operandos (usando `movb` para 8 bits, `movw` para 16, `movl` para 32 y `movq` para 64), pero esto puede ser inferido directamente del tipo de las variables, y entonces se puede escribir `x = y` para todos estos casos. Esta inferencia hecha por el compilador es inequívoca, en el sentido en que hay exactamente una respuesta posible; todas las decisiones automatizadas son de esta naturaleza, en el resto de los casos se devuelve un error.

Otra importante ventaja de Jasmin es que no es necesario trabajar con registros explícitos ni llevar la cuenta de qué registros están vivos en qué momento, siempre y cuando exista una manera de mapear todas las variables a los registros de la arquitectura. Así, por ejemplo, se puede tener un programa válido con 17 variables `reg U64`, a pesar de que solo existan 16 registros, siempre y cuando no se necesite el valor de todas al mismo tiempo.

Por último, la semántica de Jasmin está formalmente especificada. Dado un programa Jasmin y una memoria inicial, se puede ejecutar el programa y observar la memoria resultante; así es mucho más sencillo testear y validar una implementación.

3.2. El compilador

El compilador de Jasmin consta de alrededor de 84000 líneas de código, 70 % de las cuales son en Coq. Su tarea es, resumidamente: eliminar arreglos, parámetros,

```
<prog> ::= <topdef>+

<topdef> ::= <parameterdef> | <functiondef>

<parameterdef> ::= param <type> <var> = <expr>

<functiondef> ::=
  <fun_annot>
  fn <ident>(<var_decl>, ..., <var_decl>) -> (<stype>, ..., <stype>)
  {
    <var_decl>; ...; <var_decl>;
    <instr>*
    return (<var>, ..., <var>);
  }

<instr> ::=
  | <lval> = (<type>) <expr>;  // Asignacion
  | <lval>, ..., <lval> = #<ident>(<expr>, ..., <expr>);  // Assembly
  | if (<expr>) { <instr>* } else { <instr>* }  // Condicional
  | while { <instr>* } (<expr>) { <instr>* }  // Bucle
  | for <var> = <expr> to <expr> { <instr>* }  // Bucle desenrollado
  | <lval>, ..., <lval> = <ident>(<expr>, ..., <expr>);  // Funcion

<expr> ::=
  | <int>
  | <bool>
  | <var>
  | (<wsize>)[<var> + <expr>]
  | <op1> <expr>
  | <expr> <op2> <expr>

<lval> ::=
  | _  // Descartar el resultado
  | <var>  // Variable
  | (<wsize>)[<var> + <expr>]  // Valor en memoria

<fun_annot> ::= export | inline

<op1> ::= ! | -

<op2> ::= + | - | * | == | != | ...

<var_decl> ::= <stype> <var>

<stype> ::= <storage> <type>

<storage> ::= reg | stack | inline

<type> ::= int | bool | word <wsize>

<wsize> ::= U8 | U16 | U32 | U64 | U128 | U256
```

Figura 3.2: Sintaxis simplificada del lenguaje Jasmin.

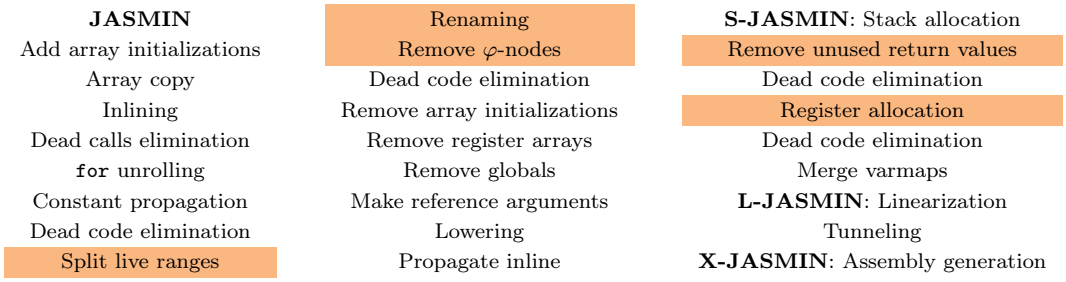


Figura 3.3: Fases del compilador.

funciones `inline` y bucles `for`; reemplazar asignaciones por operaciones *assembly*; asignar direcciones en la pila a las variables `stack` y parámetros; asignar registros a las variables `reg`; reemplazar las estructuras de control por `GOTO`s; y emitir *assembly* válido. Para lograr esto, se divide en aproximadamente 30 etapas, algunas de ellas repetidas. El programa se transforma a lo largo de estas etapas en cuatro lenguajes distintos, desde la sintaxis Jasmin presentada anteriormente hasta un lenguaje *assembly* genérico. Las fases pueden verse en la Figura 3.3.

El compilador tiene fases no verificadas (no mostradas en la figura: *parsing*, *typing*, y *pretty-printing*), fases implementadas en OCaml y verificadas en Coq (fondo de color), fases implementadas y verificadas en Coq (las demás). En mayúscula se muestran los lenguajes internos del compilador. Estos son JASMIN, que consiste en una sintaxis similar a la de la Figura 3.2 y una semántica abstracta *big-step*; S-JASMIN, con la misma sintaxis y semántica similar pero con una noción de pila; L-JASMIN, donde la única instrucción de flujo de control es `GOTO` y la semántica se define como la evolución *small-step* de una máquina abstracta; y X-JASMIN, un lenguaje *assembly* genérico. Estos lenguajes se estudiarán más en detalle en la Subsección 3.2.1. A continuación se describen brevemente algunas fases del compilador y su propósito.

Inlining. En esta fase se reemplaza las llamadas a funciones `inline` por el cuerpo de las mismas, haciendo las asignaciones correspondientes para sus argumentos y resultados. Así, x_0`, ...,` x_{n-1} `=` f`(`e_0`, ...,` e_{m-1}`)` se convierte en

arg_0 `=` e_0`; ...;` arg_{m-1} `=` e_{m-1}`;`
$f_{\texttt{body}}$
x_0 `=` res_0`; ...;` x_{n-1} `=` res_{n-1}`;`

donde $\text{arg}_0, \ldots, \text{arg}_{m-1}$ son los nombres de los argumentos y $\text{res}_0, \ldots, \text{res}_{n-1}$ los nombres de los resultados de f, y $f_{\texttt{body}}$ es el cuerpo de esta función.

Dead calls elimination. En esta fase se eliminan las funciones que nunca se llaman, principalmente las funciones `inline` que no se necesitan luego de la fase anterior.

For unrolling. En esta fase se eliminan los bucles `for` desenrollándolos. Es por esto que el rango de un bucle `for` debe poder resolverse en tiempo de compilación. Así,

`for` i `=` e `to` e' `{` c `}` se convierte en

$$
\begin{aligned}
&i \texttt{ = } e\texttt{;}\ c\\
&i \texttt{ = } e+1\texttt{;}\ c\\
&\ldots\\
&i \texttt{ = } e'\texttt{;}\ c
\end{aligned}
$$

y desde esta fase en adelante se tiene que no hay bucles `for` en los programas. No es necesario que las expresiones del rango del bucle sean constantes: el compilador ejecuta esta y las siguientes dos fases repetidamente hasta que el programa deja de cambiar, o se llega a un límite de intentos configurado por el usuario y se falla.

Constant propagation. En esta fase se reemplazan todas las variables y parámetros `int` y `bool` que tengan valor constante y se simplifican las expresiones que los involucren. Así, si se conoce que el valor de `i` es 1, `x = i + 4` se convierte en `x = 5`, y la instrucción `if (1 = 2 - 1) {` c `} else {` c' `}` se convierte en c. Notar que como la variable de un bucle `for` debe ser un `int`, esta fase hace que las asignaciones introducidas por la fase anterior sean innecesarias, transformando el código de ese ejemplo en

$$
\begin{aligned}
&i \texttt{ = } e\texttt{;}\ \ c[i := e]\\
&i \texttt{ = } e+1\texttt{;}\ \ c[i := e+1]\\
&\ldots\\
&i \texttt{ = } e'\texttt{;}\ \ c[i := e']
\end{aligned}
$$

y haciendo que el valor de i nunca sea leído. Así, la variable i y sus asignaciones pueden eliminarse en la fase siguente.

Dead code elimination. En esta fase se eliminan todas las asignaciones a variables que no son leídas después ni tienen efectos secundarios (es decir, que no están en memoria). También se eliminan las instrucciones de la forma x `=` x introducidas por el compilador, por ejemplo por la fase Inlining en el caso en que e_i sea arg_i o x_i sea res_i para algún i.

Lowering. En esta fase se convierten las asignaciones en operaciones *assembly*. Así, la instrucción `x = y;` se convierte en `x = #MOV`s`(y)` (donde s depende del tamaño de `x` e `y`), la instrucción `x += 1;` se convierte en `x = #INC(x)` y la instrucción `x = y if c;` se convierte en `x = #CMOV(y, c)`. Esta fase también inserta código para calcular las *flags* de condiciones, es decir, convierte `if (x == y) {`c_0`} else {`c_1`}` en

```
OF, CF, SF, ZF = #CMP(x, y);
if (ZF) {c0} else {c1}
```

donde `ZF` es la *flag* que determina si los dos argumentos de `CMP` son iguales (llamada *flag* cero). Esta transformación solo se hace cuando la condición se puede expresar con una sola instrucción `CMP`, en caso contrario se termina con un error.

Stack allocation. En esta fase se agrega la noción de pila a los programas. Se analizan todas las funciones del programa para calcular su uso de la pila, y se las anota con esta información. Todas las variables `stack` pasan a ser direcciones de memoria (en realidad desplazamientos desde el inicio del *stack frame*): se reemplazan todas las lecturas de estas variables por lecturas de memoria, y todas las escrituras por escrituras a memoria. Por ejemplo, se convierte el programa de la izquierda en el de la derecha, donde n es un entero:

```
stack u64 x;
x = e;
y = x;
```

```
[rsp + n] = e;
y = [rsp + n];
```

Aquí se computa cuánta memoria necesitará cada función, y el programa en general, y la semántica del lenguaje toma esto en consideración. Parte del teorema de corrección de esta fase asegura que si se cuenta con dicha cantidad de memoria entonces el programa puede ejecutarse de manera correcta y segura.

Remove unused return values. En esta fase se eliminan los resultados no utilizados de llamadas a funciones no `export`. La utilidad de esta fase está en que en desarrollos criptográficos a veces se implementan rutinas que calculan dos (o más) valores al mismo tiempo (similar a como la división euclídea calcula el cociente y el resto simultáneamente), pero dependiendo de la aplicación de la rutina puede que uno nunca se use. La modificación no puede hacerse a funciones `export` pues su signatura puede estar especificada en una API o dada por código de terceros. Consiste en un análisis de todo el programa en el que se buscan todas las llamadas de cada función, y si uno de sus resultados nunca es utilizado entonces se lo remueve de la signatura de la función y de su `return`. El análisis está implementado en OCaml, pero la modificación del código en Coq; así, la modificación del código chequea que el resultado de verdad sea descartado en todas las llamadas, permitiendo probar la corrección de esta fase.

Register allocation. En esta fase se crea una asignación de las variables locales de cada función a los registros de la arquitectura. Hay varias restricciones que una asignación válida debe cumplir, por ejemplo se debe tener en cuenta la convención de llamada de la arquitectura (qué registros se pueden utilizar para pasar argumentos), y además, naturalmente, preservar la semántica de cada función. En este momento las funciones dejan de tener variables locales, pues todas utilizan los registros y *flags* de la arquitectura, condición necesaria para la siguiente conversión de lenguaje, a L-JASMIN, donde todas las variables son globales.

Encontrar una asignación con estas características es un conocido problema NP-completo, que se suele resolver con una reducción a coloreo de grafos. Se sigue que para que la fase sea polinomial, debe ser incompleta: a veces existe una asignación válida de variables locales a registros que el compilador no encuentra. Es por esto que la fase está implementada en OCaml, pues se usan estructuras de datos y patrones de programación avanzados, combinados con heurísticas, que serían muy engorrosos de codificar eficientemente en Coq.

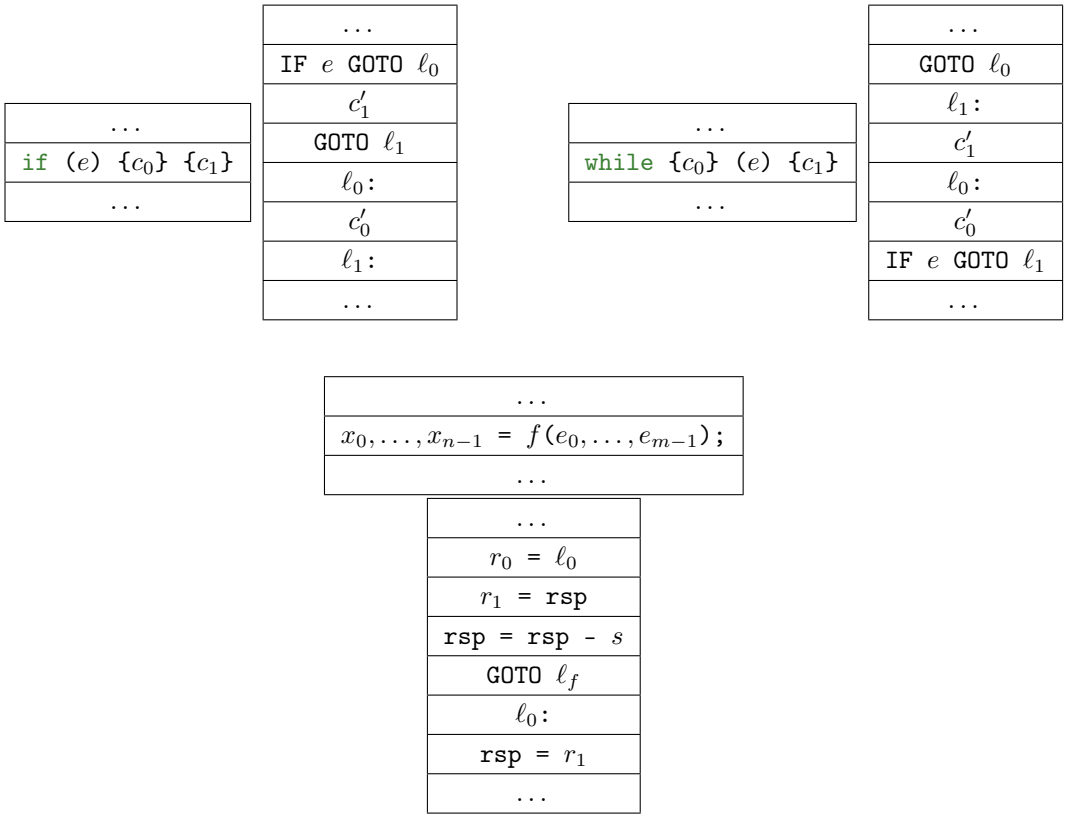


Figura 3.4: Conversión de estructuras de control de alto nivel en GOTOs.

Merge varmaps. Esta fase no modifica el programa, sino que hace un análisis estático para asegurar la validez de la asignación creada en la fase anterior. Su propósito es codificar en Coq las propiedades de la asignación encontrada, para poder probar la preservación de la semántica.

Linearization. En esta fase se convierte un programa S-JASMIN en uno L-JASMIN: se convierten todas las estructuras de control de alto nivel (`while`, `if`, llamadas a función) en GOTOs. En la semántica de este lenguaje las funciones no tienen variables locales, sino que todas las variables del programa son globales (y, gracias a Register allocation, se corresponden con los registros de la arquitectura).

En la Figura 3.4 se muestran las transformaciones de las estructuras de control en GOTOs. Se indican con una tilde (por ejemplo c'_0) al código resultante de aplicar esta misma transformación (a c_0 en el ejemplo) y con la letra ℓ a etiquetas. En el caso de la llamada a función, se utilizan variables auxiliares r_0 y r_1, donde se almacenan la etiqueta a la que saltar al terminar la llamada y el tope de la pila respectivamente. Además, se deja espacio para *stack frame* de la llamada, cuyo tamaño se denomina s en la Figura.

Tunneling. Una de las desventajas de no tener acceso a instrucciones de control al estilo `GOTO` es que ciertos patrones de programación se compilan ineficientemente, por ejemplo cuando un `if` sigue a otro puede pasar que haya un salto a otra instrucción de salto. En esta fase se resuelven estas situaciones: si se detecta en el programa la instrucción `GOTO` ℓ y además la secuencia ℓ`:` `GOTO` ℓ', se reemplaza la primer instrucción por `GOTO` ℓ'.

Assembly generation. En esta fase se convierte un programa L-JASMIN en uno X-JASMIN. En la siguiente subsección se detallan las características de este lenguaje, pero en esencia es L-JASMIN con varias restricciones; por ejemplo, las condiciones de los saltos deben ser combinaciones de *flags* aceptadas por la arquitectura. Este es el último lenguaje interno del compilador, sobre el cual se define la fase de *pretty-printing* que produce el código *assembly* final.

3.2.1. Lenguajes del compilador

Para lograr su propósito de transformar programas Jasmin en código *assembly*, el compilador de Jasmin se divide en las fases mencionadas anteriormente que transforman un programa a lo largo de varias representaciones intermedias, cada vez más parecidas a *assembly*.

Una propiedad muy importante del compilador es que preserve la semántica del programa, es decir, que el comportamiento del programa Jasmin y el comportamiento del código *assembly* generado sean equivalentes. Para lograr esto, se usa una estrategia composicional: cada fase del compilador preserva la semántica del programa, y así, transitivamente, la composición de todas estas fases también la preserva. Es entonces necesario que cada representación intermedia del programa tenga una semántica definida rigurosamente.

En esta subsección se describen de manera simplificada la sintaxis, características y semántica de los cuatro lenguajes internos del compilador de Jasmin: JASMIN, S-JASMIN, L-JASMIN y X-JASMIN.

El lenguaje JASMIN

El primer lenguaje utilizado en el compilador es JASMIN, una codificación directa de `<instr>` de la Figura 3.2. El código JASMIN es una lista de estas instrucciones.

Un programa en este contexto, `prog` en Coq, es una lista de nombres y definiciones de función, como se ve en la Figura 3.5. La definición de una función consiste en los tipos y nombres de sus argumentos, los tipos y nombres de sus resultados, el cuerpo de la función, e información extra. Al principio de la compilación no hay información extra, pero las fases siguientes utilizarán este campo para almacenar datos sobre las funciones, como por ejemplo cuánta memoria se necesita para su pila.

La semántica del lenguaje JASMIN se construye en base a una relación entre dos estados y una instrucción, como se ve en la Figura 3.7. Se denotará $[\![\_, \_]\!]_{\mathtt{i}} \Downarrow_{\mathtt{p}} \_$ a esta relación $\mathtt{state} \times \mathtt{instr} \times \mathtt{state}$, donde `p` es un programa fijo (en lo que sigue a

```
Record fdef :=
  {
    f_tin : seq type;
    f_params : seq var;
    f_tout : seq type;
    f_res : seq var;
    f_body : seq instr;
    f_extra : fun_extra;
  }.
```

```
Record prog :=
  {
    p_funcs : seq (fname * fdef);
  }.
```

Figura 3.5: Definición de `prog`.

```
value ::= <bool> | <int> | <word> | Vundef <type>
```

Figura 3.6: Definición de los valores del lenguaje JASMIN.

veces se omite por simplicidad) y los estados consisten en una memoria y un mapa de variables. Si bien la definición precisa de una memoria es bastante compleja, para los fines de presentar la semántica es suficiente definirla como una función parcial de direcciones en *bytes* (es decir `word U64 -> option (word U8)`). Los mapas de variables son funciones parciales de variables en valores, que son booleanos, enteros, o palabras, como puede verse en la Figura 3.6.

Se definen también juicios auxiliares para las operaciones, y también para escribir, leer y evaluar expresiones en un estado. El juicio $[\![\mathsf{args}, \mathsf{op}]\!]_{\mathsf{op}} \Downarrow \mathsf{res}$ expresa que la operación *assembly* op con valores args como argumentos produce res como resultado; esto es parte de la definición de cada instrucción de la arquitectura a la que se compila y se describe en más detalle en la Subsección 4.2.1. El juicio $\mathsf{vm}[\mathsf{x}] \Downarrow \mathsf{v}$ significa que leer la variable x del mapa de variables vm resulta exitosamente en v. Por otro lado, el juicio $(\mathsf{ty})\mathsf{v} \Downarrow \mathsf{v}'$ representa el *truncamiento* de valores, y afirma que v es una palabra de n bits, ty es `word` m con $m \leq n$ y v' los m *bits* menos significativos de v. La escritura de valores se denota $\mathsf{s}[\mathsf{lv} \leftarrow (\mathsf{ty})\mathsf{v}] \Downarrow \mathsf{s}'$, afirmando que se puede truncar el valor v al tipo ty y que escribir ese valor en lv en el estado s resulta exitosamente en s'. Finalmente, que la evaluación de una expresión (`<expr>` en la Figura 3.2) e en un estado s sea exitosa y resulte en un valor v se expresa con el juicio $[\![\mathsf{s}, \mathsf{e}]\!]_{\mathsf{e}} \Downarrow \mathsf{v}$. Toda notación se extiende a listas de elementos (por ejemplo $\mathsf{vm}[\mathsf{x}_0, \ldots, \mathsf{x}_n] \Downarrow \mathsf{v}_0, \ldots, \mathsf{v}_n$ indica que $\mathsf{vm}[\mathsf{x}_i] \Downarrow \mathsf{v}_i$ para cada i), y suele abusarse usando estados donde se esperan memorias o mapa de variables (indicando que se usa la memoria o el mapa de ese estado).

El lenguaje S-JASMIN

El lenguaje S-JASMIN tiene la misma sintaxis que JASMIN, salvo que no hay bucles `for`. Además, la semántica también es muy parecida, con la salvedad de que no

$$\frac{}{[\![\, s, [\,] \,]\!]_c \Downarrow s} \qquad \frac{[\![\, s, i \,]\!]_i \Downarrow s' \qquad [\![\, s', c \,]\!]_c \Downarrow s''}{[\![\, s, i :: c \,]\!]_c \Downarrow s''}$$

$$\frac{[\![\, s, e \,]\!]_e \Downarrow v \qquad s[\, lv \leftarrow (ty)v \,] \Downarrow s'}{[\![\, s, lv \ =(ty)\ e; \,]\!]_i \Downarrow s'} \qquad \frac{[\![\, s, e \,]\!]_e \Downarrow \texttt{true} \qquad [\![\, s, c_0 \,]\!]_c \Downarrow s'}{[\![\, s, \texttt{if (e) \{}c_0\texttt{\} \{}c_1\texttt{\}} \,]\!]_i \Downarrow s'}$$

$$\frac{\begin{array}{c}[\![\, s, c_0 \,]\!]_c \Downarrow s' \\ [\![\, s', e \,]\!]_e \Downarrow \texttt{true} \qquad [\![\, s', c_1 \,]\!]_c \Downarrow s'' \\ [\![\, s'', \texttt{while \{}c_0\texttt{\} (e) \{}c_1\texttt{\}} \,]\!]_i \Downarrow s'''\end{array}}{[\![\, s, \texttt{while \{}c_0\texttt{\} (e) \{}c_1\texttt{\}} \,]\!]_i \Downarrow s'''} \qquad \frac{[\![\, s, c_0 \,]\!]_c \Downarrow s' \qquad [\![\, s', e \,]\!]_e \Downarrow \texttt{false}}{[\![\, s, \texttt{while \{}c_0\texttt{\} (e) \{}c_1\texttt{\}} \,]\!]_i \Downarrow s'}$$

$$\frac{\begin{array}{c}[\![\, s, es \,]\!]_e \Downarrow args \\ [\![\, args, op \,]\!]_{op} \Downarrow res \\ s[\, lvs \leftarrow res \,] \Downarrow s'\end{array}}{[\![\, s, lvs \ = \ \#op(es)\,; \,]\!]_i \Downarrow s'} \qquad \frac{\begin{array}{c}[\![\, m, vm, es \,]\!]_e \Downarrow args \\ [\![\, fn, m, args \,]\!]_{call} \Downarrow (m', res) \\ (m', vm)[\, lvs \leftarrow res \,] \Downarrow s''\end{array}}{[\![\, m, vm, lvs \ = \ fn(es)\,; \,]\!]_i \Downarrow s''}$$

$$\frac{\begin{array}{c}[\![\, s, lo \,]\!]_e \Downarrow vlo \qquad [\![\, s, hi \,]\!]_e \Downarrow vhi \\ [\![\, s, x, [vlo, \ldots, vhi], c \,]\!]_{for} \Downarrow s'\end{array}}{[\![\, s, \texttt{for x = lo to hi \{c\}} \,]\!]_i \Downarrow s'} \qquad \frac{}{[\![\, s, x, [\,], c \,]\!]_{for} \Downarrow s} \qquad \frac{\begin{array}{c}s[\, x \leftarrow n \,] \Downarrow s' \qquad [\![\, s', c \,]\!]_c \Downarrow s'' \\ [\![\, s'', x, ns, c \,]\!]_{for} \Downarrow s'''\end{array}}{[\![\, s, x, n :: ns, c \,]\!]_{for} \Downarrow s'''}$$

$$\frac{\begin{array}{c}(fn, fd) \in \texttt{p\_funcs p} \\ (m, \varnothing)[\, \texttt{f\_params fd} \leftarrow (\texttt{f\_tin fd})args \,] \Downarrow s \\ [\![\, s, \texttt{f\_body fd} \,]\!]_c \Downarrow (m', vm') \\ vm'[\, \texttt{f\_res fd} \,] \Downarrow vs \\ (\texttt{f\_tout fd})vs \Downarrow res\end{array}}{[\![\, fn, m, args \,]\!]_{call} \Downarrow (m', res)}$$

Figura 3.7: Semántica del lenguaje JASMIN.

$$\frac{\begin{array}{c}(\mathsf{fn},\mathsf{fd}) \in \texttt{p\_funcs}\ \mathsf{p} \\ \texttt{stk-initialize}\ \mathsf{m} \Downarrow \mathsf{s} \\ \mathsf{s}[\,\texttt{f\_params}\ \mathsf{fd} \leftarrow (\texttt{f\_tin}\ \mathsf{fd})\mathsf{args}\,] \Downarrow \mathsf{s}' \\ [\![\,\mathsf{s}', \texttt{f\_body}\ \mathsf{fd}\,]\!]_{\texttt{c}} \Downarrow (\mathsf{m}', \mathsf{vm}') \\ \texttt{stk-finalize}\ \mathsf{m}' \Downarrow \mathsf{m}'' \\ \mathsf{vm}'[\,\texttt{f\_res}\ \mathsf{fd}\,] \Downarrow \mathsf{vs} \\ (\texttt{f\_tout}\ \mathsf{fd})\mathsf{vs} \Downarrow \mathsf{res}\end{array}}{[\![\,\mathsf{fn}, \mathsf{m}, \mathsf{args}\,]\!]_{\texttt{call}} \Downarrow (\mathsf{m}'', \mathsf{res})}$$

Figura 3.8: Semántica de llamadas a funciones en S-JASMIN.

son necesarias las reglas para `for` y que $[\![\ \_, \_, \_\ ]\!]_{\texttt{call}} \Downarrow_{\_}\ (\_, \_)$ se generaliza como se ve en la Figura 3.8. Los programas ahora además llevan información sobre el tamaño de la pila que cada función necesita, si deben pasar argumentos mediante la misma, y cómo se almacena el *return pointer* (en un registro o en la pila). Debido a estas similaridades, no se tienen dos definiciones separadas para JASMIN y S-JASMIN, sino que se tiene una sola parametrizada por las diferencias. Por un lado, desde la fase `for` unrolling se mantiene el invariante de que no hay bucles `for` en el programa. Y por otro, las funciones `stk-initialize` y `stk-finalize` preparan y liberan espacio en la pila para la función: En JASMIN `stk-initialize` es simplemente $\mathsf{m} \mapsto (\mathsf{m}, \varnothing)$ y `stk-finalize` la identidad, mientras que en S-JASMIN la primera crea un *stack frame* y la segunda lo libera.

El lenguaje L-JASMIN

El propósito de este lenguaje es tener una representación de los programas a bajo nivel sin restricciones particulares de arquitecturas reales, como manejar registros, *flags* y condiciones.

La sintaxis del lenguaje L-JASMIN es la de la Figura 3.9. En este lenguaje la única instrucción de flujo de control es el `GOTO` condicional, con el que se puede saltar a etiquetas en el código. Estas etiquetas son abstractas, el único requerimiento es que sean isomorfas a los punteros (es decir, que existen funciones inversas $\lfloor\_\rfloor$ y $\lceil\_\rceil$ que llevan etiquetas a palabras de 64 bits y viceversa).

La semántica, por otro lado, está dada por una relación entre estados que se define basándose en la semántica de instrucciones, como se ve en la Figura 3.10. Un estado consiste en una memoria, un mapa de variables, un nombre de función y el índice de la próxima instrucción a ejecutar; mientras que un programa es una lista de nombres de función y listas de instrucciones, similarmente al caso de JASMIN. La semántica del lenguaje se define como la clausura reflexiva transitiva de la relación que representa un paso de ejecución, denotada $[\![\ \_\ ]\!]_{\texttt{lsem1}} \Downarrow \_$, la cual consiste en

```
<linstr> ::=
  | <lval>, ..., <lval> = #<ident>(<expr>, ..., <expr>);  // Assembly
  | <label>:  // Etiqueta
  | <lval> = <label>;  // Almacenar etiqueta
  | IF <expr> GOTO <expr>;  // Salto condicional
```

Figura 3.9: Sintaxis del lenguaje L-JASMIN.

buscar la definición de la función a ejecutar (cuyo nombre, fn, se guarda en el estado), la próxima instrucción a ejecutar (cuyo índice, pc, también es parte del estado), y ejecutar dicha instrucción usando $[\![\_, \_]\!]_{\texttt{li}} \Downarrow \_$. La semántica de `GOTO` en el caso en el que la condición sea verdadera busca en el programa la etiqueta correspondiente, determinando así el nombre de la función y el índice de la instrucción a donde debe hacerse el salto (`lookup` en la Figura).

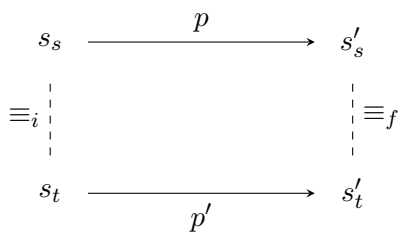
El lenguaje X-JASMIN

La sintaxis del lenguaje X-JASMIN es básicamente la de L-JASMIN con varias restricciones. Los argumentos de las instrucciones *asssembly* ya no son expresiones sino registros, direcciones de memoria, inmediatos, condiciones, etc; y las etiquetas se deben almacenar en registros. La condición de los `GOTO`s debe ser una condición aceptada por la arquitectura (como `EQ` o `LT`) y la segunda expresión debe ser o una etiqueta o un registro (que contiene la dirección a la que saltar).

En este lenguaje se diferencian los distintos tipos de saltos para hacer el *pretty-printing* más directo: saltos absolutos (a una dirección inmediata), indirectos (mediante un registro), `CALL`s y `RET`s (para funciones).

3.3. Verificación del compilador

Cuando se dice que el compilador de Jasmin está verificado se está diciendo que la semántica del programa de entrada al compilador es equivalente a la del *assembly* de salida. La semántica de todos los lenguajes involucrados está dada como transformaciones de estados. De manera abstracta, se busca asegurar que si p es el programa de entrada y partiendo de un estado s_s ($\__s$ por *source*) llega a un estado s'_s, entonces la salida del compilador p' lleva a cualquier estado s_t ($\__t$ por *target*) equivalente a s_s a un estado s'_t, que es equivalente a s'_s



$$\frac{(\mathsf{fn}, \mathsf{fd}) \in \texttt{lp\_funcs p} \qquad (\texttt{lf\_body fd})\,[\mathsf{pc}] = \mathsf{i} \qquad [\![\, \mathsf{m}, \mathsf{vm}, \mathsf{fn}, \mathsf{pc}, \mathsf{i} \,]\!]_{\texttt{li}} \Downarrow \mathsf{s}'}{[\![\, \mathsf{m}, \mathsf{vm}, \mathsf{fn}, \mathsf{pc} \,]\!]_{\texttt{lsem1}} \Downarrow \mathsf{s}'}$$

$$\frac{[\![\, \mathsf{m}, \mathsf{vm}, \mathsf{es} \,]\!]_{\mathsf{e}} \Downarrow \mathsf{args} \qquad [\![\, \mathsf{args}, \mathsf{op} \,]\!]_{\mathsf{op}} \Downarrow \mathsf{res} \qquad (\mathsf{m}, \mathsf{vm})[\,\mathsf{lvs} \leftarrow \mathsf{res}\,] \Downarrow (\mathsf{m}', \mathsf{vm}')}{[\![\, \mathsf{m}, \mathsf{vm}, \mathsf{fn}, \mathsf{pc}, \mathsf{lvs} \texttt{ = \#op(es) ;} \,]\!]_{\texttt{li}} \Downarrow (\mathsf{m}', \mathsf{vm}', \mathsf{fn}, \mathsf{pc} + 1)}$$

$$\frac{}{[\![\, \mathsf{m}, \mathsf{vm}, \mathsf{fn}, \mathsf{pc}, \mathsf{lbl}\texttt{:} \,]\!]_{\texttt{li}} \Downarrow (\mathsf{m}, \mathsf{vm}, \mathsf{fn}, \mathsf{pc} + 1)}$$

$$\frac{(\mathsf{m}, \mathsf{vm})[\,\mathsf{lv} \leftarrow \lfloor \mathsf{lbl} \rfloor\,] \Downarrow (\mathsf{m}', \mathsf{vm}')}{[\![\, \mathsf{m}, \mathsf{vm}, \mathsf{fn}, \mathsf{pc}, \mathsf{lv} \texttt{ = } \mathsf{lbl} \,]\!]_{\texttt{li}} \Downarrow (\mathsf{m}', \mathsf{vm}', \mathsf{fn}, \mathsf{pc} + 1)}$$

$$\frac{[\![\, \mathsf{m}, \mathsf{vm}, \mathsf{c} \,]\!]_{\mathsf{e}} \Downarrow \texttt{true} \qquad \lceil [\![\, \mathsf{m}, \mathsf{vm}, \mathsf{e} \,]\!]_{\mathsf{e}} \rceil \Downarrow \mathsf{lbl} \qquad \texttt{lookup p lbl} = (\mathsf{fn}', \mathsf{pc}')}{[\![\, \mathsf{m}, \mathsf{vm}, \mathsf{fn}, \mathsf{pc}, \texttt{IF c GOTO e} \,]\!]_{\texttt{li}} \Downarrow (\mathsf{m}, \mathsf{vm}, \mathsf{fn}', \mathsf{pc}' + 1)}$$

$$\frac{[\![\, \mathsf{m}, \mathsf{vm}, \mathsf{c} \,]\!]_{\mathsf{e}} \Downarrow \texttt{false}}{[\![\, \mathsf{m}, \mathsf{vm}, \mathsf{fn}, \mathsf{pc}, \texttt{IF c GOTO e} \,]\!]_{\texttt{li}} \Downarrow (\mathsf{m}, \mathsf{vm}, \mathsf{fn}, \mathsf{pc} + 1)}$$

Figura 3.10: Semántica del lenguaje L-JASMIN.

o en símbolos

$$\forall s_s,\, s'_s,\, s_t.\, [\![\, s_s \,]\!]_s \Downarrow_p s'_s \wedge s_s \equiv_i s_t \implies \exists s'_t.\, [\![\, s_t \,]\!]_t \Downarrow_{p'} s'_t \wedge s'_s \equiv_f s'_t$$

El significado del teorema de corrección del compilador entonces depende radicalmente de cómo se definan «llevar un estado a otro» (denotado antes por $[\![\ \_\ ]\!]_\_ \Downarrow_\_ \_$), la relación de equivalencia entre estados iniciales ($\_ \equiv_i \_$) y entre estados finales ($\_ \equiv_f \_$). Estas definiciones tienen varias sutilezas.

Para el lenguaje JASMIN por ejemplo, un programa consiste en una serie de definiciones de funciones, por lo que la ejecución de un programa refiere en realidad a la de alguna de estas funciones, y se busca que la propiedad de preservación de la semántica se cumpla para todas las funciones definidas. Los demás lenguajes del compilador también tienen una noción de «definición de función» o de «bloque de código con nombre», así que la propiedad de corrección suele ser que para toda definición de función f en p, existe una defincón de función f' en p' tal que

$$\forall s_s,\, s'_s,\, s_t.\, [\![\, f,\, s_s \,]\!]_s \Downarrow_p s'_s \wedge s_s \equiv s_t \implies \exists s'_t.\, [\![\, f',\, s_t \,]\!]_t \Downarrow_{p'} s'_t \wedge s'_s \equiv s'_t$$

Por otro lado, la definición de la equivalencia de estados puede ser bastante compleja ya que los estados del lenguaje fuente s_s pueden ser muy distintos a los del lenguaje de resultado s_t. Por ejemplo, un estado en el contexto de una llamada a función en JASMIN consiste de una memoria y una lista de valores, mientras que en X-JASMIN consiste en una memoria y un banco de registros. Más adelante se precisarán estas relaciones.

Así, para la verificación del compilador se usan juicios lógicos que refieren a la semántica de una función en un programa. Para los lenguajes JASMIN y S-JASMIN se definen

$$[\![\, \mathsf{f}, \mathsf{m_J}, \mathsf{args} \,]\!]_{\mathtt{J}} \Downarrow_{\mathsf{p}} (\mathsf{m'_J}, \mathsf{res}) \triangleq [\![\, \mathsf{f}, \mathsf{m_J}, \mathsf{args} \,]\!]_{\mathtt{call}} \Downarrow_{\mathsf{p}} (\mathsf{m'_J}, \mathsf{res})$$
$$[\![\, \mathsf{f}, \mathsf{m_S}, \mathsf{args} \,]\!]_{\mathtt{S}} \Downarrow_{\mathsf{p}} (\mathsf{m'_S}, \mathsf{res}) \triangleq [\![\, \mathsf{f}, \mathsf{m_S}, \mathsf{args} \,]\!]_{\mathtt{call}} \Downarrow_{\mathsf{p}} (\mathsf{m'_S}, \mathsf{res})$$

con las definciones de $[\![\ \_, \_, \_\ ]\!]_{\mathtt{call}} \Downarrow_\_ (\_, \_)$ dadas anteriormente para cada lenguaje. En ambos casos entonces los estados son memorias y listas de valores. Una distinción que antes se omitió es que las memorias no tienen la misma estructura a lo largo de todas las fases. Si bien siempre son esencialmente una función parcial de direcciones a *bytes*, se decoran con diferentes datos adicionales a medida que se refina el programa, como límites para la pila, *frames* para las funciones, etc. Las diferencias concretas no son muy importantes, pero señalaremos los distintos tipos de memoria como $\mathsf{m_J}$, $\mathsf{m_S}$, $\mathsf{m_L}$ y $\mathsf{m_X}$.

El juicio usado para L-JASMIN se define como

$$[\![\, \mathsf{f}, \mathsf{m_L}, \mathsf{vm} \,]\!]_{\mathtt{L}} \Downarrow_{\mathsf{p}} (\mathsf{m'_L}, \mathsf{vm'}) \triangleq [\![\, \mathsf{m_L}, \mathsf{vm}, \mathsf{f}, 0 \,]\!]_{\mathtt{lsem1}^*} \Downarrow_{\mathsf{p}} (\mathsf{m'_L}, \mathsf{vm'}, \mathsf{f}, |\mathtt{lf\_body}\ \mathsf{f}|)$$

donde se usa la clausura reflexiva transitiva de la relación introducida en 3.10. Para X-JASMIN se usa un juicio similar al anterior.

Cada fase del compilador tiene una prueba de que preserva la semántica del programa. Como se mencionó, hay varias versiones de esta propiedad, dependiendo de las definciones de $\Downarrow$ y $\equiv$, y estas pruebas de corrección de fase se componen para construir una del compilador completo. Para las fases anteriores a Linearization hay tres versiones similares y bastante directas, ya que lenguaje de entrada y salida de estas fases son el mismo, JASMIN, o muy similares, JASMIN y S-JASMIN (en el caso de Stack allocation). La más restrictiva expresa que para toda función `export` cuyo nombre sea f, si el resultado de la fase en el programa p es p', se tiene que

$$[\![\mathsf{f}, \mathsf{m}_\mathtt{J}, \mathsf{args}]\!]_\mathtt{J} \Downarrow_\mathsf{p} (\mathsf{m}'_\mathtt{J}, \mathsf{res}) \implies [\![\mathsf{f}, \mathsf{m}_\mathtt{J}, \mathsf{args}]\!]_\mathtt{J} \Downarrow_{\mathsf{p}'} (\mathsf{m}'_\mathtt{J}, \mathsf{res})$$

es decir, que el resultado y el comportamiento sobre la memoria de la función son exactamente los mismos. La equivalencia de estados es entonces la igualdad.

La segunda versión es usada para la mayoría de estas fases, y expresa algo similar a la anterior con la salvedad de que el resultado de llamar a una función en el programa original debe «estar incluído» en el de llamarla en el programa compilado p'

$$[\![\mathsf{f}, \mathsf{m}_\mathtt{J}, \mathsf{args}]\!]_\mathtt{J} \Downarrow_\mathsf{p} (\mathsf{m}'_\mathtt{J}, \mathsf{res}_s) \implies \exists \mathsf{res}_t. [\![\mathsf{f}, \mathsf{m}_\mathtt{J}, \mathsf{args}]\!]_\mathtt{J} \Downarrow_{\mathsf{p}'} (\mathsf{m}'_\mathtt{J}, \mathsf{res}_t) \wedge \mathsf{res}_s \subseteq \mathsf{res}_t$$

En este caso la equivalencia de estados iniciales es la igualdad, y la de estados finales es la igualdad para las memorias e inclusión para los valores. La inclusión de valores está dada por el truncamiento: un valor v está incluido en otro valor v' si v es el truncamiento de v' para algún tamaño de palabra de *bits*. Así, el compilador puede utilizar un registro de 64 *bits* para almacenar un valor de 8 *bits*, siempre que el *byte* menos significativo del registro coincida con el valor a preservar.

La siguiente versión se utiliza sólo para la fase de Stack allocation, y expresa lo mismo que la anterior con una salvedad similar para la memoria. La memoria puede estar «extendida» con algunos datos

$$\begin{aligned}[\![\mathsf{f}, \mathsf{m}_\mathtt{J}, \mathsf{args}_s]\!]_\mathtt{J} \Downarrow_\mathsf{p} (\mathsf{m}'_\mathtt{J}, \mathsf{res}_s) \wedge (\mathsf{m}_\mathtt{J}, \mathsf{args}_s) \equiv_i (\mathsf{m}_\mathtt{S}, \mathsf{args}_t) \wedge \mathsf{WF}(\mathsf{m}_\mathtt{J}, \mathsf{args}_s) \\ \implies \exists \mathsf{m}'_\mathtt{S}, \mathsf{res}_t. [\![\mathsf{f}, \mathsf{m}_\mathtt{S}, \mathsf{args}_t]\!]_\mathtt{S} \Downarrow_{\mathsf{p}'} (\mathsf{m}'_\mathtt{S}, \mathsf{res}_t) \wedge (\mathsf{m}'_\mathtt{J}, \mathsf{res}_s) \equiv_f (\mathsf{m}'_\mathtt{S}, \mathsf{res}_t)\end{aligned}$$

En este caso la equivalencia de estados iniciales se define como la inclusión para memorias y valores, donde la inclusión para memorias se define de la siguiente manera

$$\forall \mathsf{x}, \mathsf{w}. \mathsf{m}_\mathtt{J}[\mathsf{x}] \Downarrow \mathsf{w} \implies \mathsf{m}_\mathtt{S}[\mathsf{x}] \Downarrow \mathsf{w}$$

y además se agrega una precondición a los estados para asegurar que estén bien formados: en este caso solamente se requiere que haya suficiente espacio en la pila para la ejecución de f

$$0 \leq \texttt{f\_stk\_max}\ \mathsf{f} \leq \texttt{stk\_top}\ \mathsf{m}_\mathtt{S} - \texttt{stk\_limit}\ \mathsf{m}_\mathtt{S}$$

donde `f_stk_max` es la cantidad de memoria requerida y se calcula en la fase Stack allocation (es parte de la información extra `f_extra` mencionada anteriormente).

Para las siguientes fases se prueban propiedades análogas, con variaciones simples de acuerdo a los lenguajes que relacionen, pues las semánticas de JASMIN y S-JASMIN ya son de un nivel bastante bajo. Las únicas diferencias son que entre S-JASMIN y L-JASMIN se deben llevar los resultados de la llamada a función al mapa de variables global, y entre L-JASMIN y X-JASMIN se debe llevar el mapa de variables global a un banco de registros. Además, se tienen en cuenta los valores de variables especiales como el tope de la pila.

El teorema de corrección del compilador es simplemente la composición de todas estas propiedades, relacionando programas JASMIN con programas X-JASMIN, por lo que se tiene la versión más débil (con inclusión en vez de igualdad de estados). La inclusión de estados también se define como una composición

$$s_{\mathtt{J}} \subseteq s_{\mathtt{X}} \triangleq \exists s_{\mathtt{S}}, s_{\mathtt{L}}.\, s_{\mathtt{J}} \subseteq s_{\mathtt{S}} \subseteq s_{\mathtt{L}} \subseteq s_{\mathtt{X}}$$

y las precondiciones deben propagarse hasta el nivel más alto.

4 Cambios al compilador

Este capítulo constituye la segunda parte del trabajo, en la que se describe una propuesta de generalización del compilador y luego una extensión para soportar una arquitectura específica. Dichas tareas fueron implementadas en Coq y en OCaml, y pueden encontrarse en el repositorio del compilador de Jasmin.

4.1. Generalizando el compilador

Si bien el compilador y el lenguaje se diseñaron en principio de manera genérica, ciertas partes de la implementación del compilador asumen caracterísiticas de la arquitectura a la que se compila. Ejemplos de esto son aspectos simples como el tamaño de los punteros (y por ende de la memoria) o la cantidad de registros disponibles, y también sutilezas como la posibilidad de copiar valores entre dos posiciones de memoria sin necesitar un registro auxiliar.

4.1.1. Generalizando punteros

La primer tarea llevada a cabo en la generalización del compilador fue considerablemente simple: se abstrajo la noción de puntero, que previamente era solo un sinónimo de palabra de 64 *bits*.

En primera instancia se agregó un parámetro `Uptr` de tipo `wsize` a todas las definiciones del compilador que involucraban punteros directa o indirectamente. Naturalmente, todas las pruebas se vieron afectadas ya que las memorias dependen de este parámetro y forman parte de la semántica de los programas en todos los lenguajes del compilador. Gracias a las facilidades de secciones y contextos en Coq (`Section` y `Context`) esta tarea fue mucho menos tediosa de lo que aparenta, y no se debió modificar todo el código del compilador. Sin embargo, no se pudo evitar una cantidad considerable de cambios meramente sintácticos. Además se tuvieron que reescribir varias pruebas, por ejemplo sobre las direcciones alcanzables con un puntero, pues se resolvían por computación o con tácticas automáticas (como `lia`) utilizando el hecho de que los punteros son isomorfos a los naturales módulo 2^{64}.

4.1.2. Generalizando instrucciones

Otra parte del compilador que debió generalizarse fue la inyección de código en algunas fases, porque claramente el código a introducir varía con la arquitectura. No solo las instrucciones concretas cambian, sino que ciertas modificaciones no pueden hacerse en todas las arquitecturas: en x86 la instrucción `MOV (%eax), (%ebx)` copia

un valor en memoria a otra dirección de memoria, pero en ARM esto no puede hacerse sin un registro auxiliar (primero debe copiarse el valor al registro con una instrucción *load* y luego escribirlo en memoria con una *store*). Una instancia en la que ocurre esto es en la fase Stack allocation, que convierte un programa JASMIN en uno S-JASMIN como se explica en la Sección 3.2, y donde se generan instrucciones que calculan la dirección de variables en la pila. Se necesita una instrucción que compute la suma del tope de la pila (almacenado en un registro) con el desplazamiento de la variable (un valor inmediato), por lo que se introduce un parámetro `mov_ofs` a esta fase

```
Context (mov_ofs : lval -> expr -> Z -> option instr).
```

que se instancia con `ADD` en ARM y con `MOV` o `LEA` en x86, por ejemplo. El parámetro es parcial ya que las instrucciones de cada arquitectura imponen ciertas restricciones sobre sus argumentos, por ejemplo un rango permitido para los inmediatos. En los casos en los que no se pueda generar una instrucción con este parámetro el compilador falla, pero como solo se usa para modificaciones generadas por el compilador mismo, esto no debería ocurrir (aunque esto no se prueba en Coq).

4.1.3. Parámetros de fase y de arquitectura

Luego de comprobar que los cambios anteriores funcionaban correctamente, se agregaron *parámetros de fase* y *parámetros de arquitectura* al compilador para contar con una manera más genérica de introducir dependencias similares y preparar el compilador para futuras generalizaciones.

Los parámetros de arquitectura son detalles específicos que se deben modelar para cada arquitectura a soportar en el compilador; un ejemplo de estos es el tamaño de los punteros antes mencionado. Los parámetros de fase son aquello que las fases del compilador utilizan para analizar o modificar el código y depende de alguna característica de la arquitectura, por ejemplo `mov_ofs` para la fase Stack allocation. La diferenciación de parámetros de fase y de arquitectura es meramente una convención, se piensa a los primeros como relacionados con una fase en específico mientras que los segundos son aspectos más generales o compartidos por varias fases distintas. Hay parámetros que sólo son utilizados en las pruebas de corrección del compilador, la mayoría de estos son pruebas de propiedades de la arquitectura.

Estos parámetros se implementaron utilizando *typeclasses* y *coerciones* para abusar el sistema de inferencia de Coq y no tener que pasar los parámetros explícitamente. Así, para los dos ejemplos anteriores se tiene

```
Class memory_params :=
  {
    Uptr : wsize;
  }.

Class stack_allocation_params :=
  {
    mov_ofs : lval -> expr -> Z -> option instr;
  }.
```

y de forma similar para cada fase o componente que necesite parámetros específicos de la arquitectura.

Una desventaja de esta estrategia es que al complejizarse el desarrollo aumenta el número de typeclasses y coerciones, y además se definen estructuras anidadas con estos parámetros, lo que tiene un impacto en el tiempo de chequeo y compilación del compilador (de la porción implementada en Coq). De hecho, en ciertas situaciones existe más de una forma de construir un parámetro implícito (debido a las coerciones) y se debe sobreescribir la prioridad de la instancia a utilizar o darla explícitamente para evitar que el compilador dé Coq de un error o, en algunos casos, consuma toda la memoria RAM y cause una falla del sistema operativo.

Se omiten algunas generalizaciones en este escrito por brevedad, como extensiones al modelado de arquitecturas y la manera de controlar las *flags* en el código fuente. También se hicieron generalizaciones en la porción OCaml del compilador, aunque muy similares y mucho más reducidas en cantidad y tamaño.

4.2. Modelando ARM

Para validar las generalizaciones anteriores se agregó soporte para una arquitectura concreta al compilador, específicamente ARM Cortex M4 [3]. Esta tarea consiste en describir la arquitectura, instanciar los parámetros de arquitectura y de fase, e implementar la fase de Lowering.

4.2.1. Instrucciones

La descripción de la arquitectura consiste en declarar los registros, las *flags*, las condiciones, y otros detalles similares. Naturalmente, también hubo que definir las instrucciones de la arquitectura, tanto para la generación de código como para proveerlas como *inline assembly*. La semántica de las instrucciones es necesaria para dar la de los programas, y por ende es una parte clave de la prueba de corrección del compilador.

La declaración de una instrucción en el compilador de Jasmin consiste en definir un *record* con información sobre la misma. Como se ve en la Figura 4.1, se dan el tipo de los argumentos y resultados, la semántica, descripciones de los argumentos de entrada y salida de la instrucción y sus posibles signaturas.

Los tipos de los argumentos y resultados pueden ser palabra de bits, entero o booleano, y se utilizan para definir el tipo de la semántica: si los tipos de los argumentos son $\sigma_1, \ldots, \sigma_n$ y los de resultados $\tau_1, \ldots, \tau_m$, el tipo de la semántica será $\sigma_1 \to \ldots \to \sigma_n \to (\tau_1 * \ldots * \tau_m + \texttt{Unit})$. Por ejemplo, la semántica de `ADD` es

```
Definition sem_ADD (wn wm : word U32) : option (word U32) :=
  Some (wn + wm).
```

Las descripciones de los argumentos de entrada y salida y la signatura se usan para verificar que las instrucciones *inline assembly* estén bien formadas y para generar el *assembly* al final de la compilación. Estas descripciones indican detalles sobre un

```
Record instr_desc :=
  {
    (* Tipo de los argumentos. *)
    id_tin : seq type;
    (* Tipo de los resultados. *)
    id_tout : seq type;
    (* Semantica. *)
    id_semi : sem_prod id_tin (option (sem_tuple id_tout));
    (* Descripcion de los argumentos de entrada. *)
    id_in : seq arg_desc;
    (* Descripcion de los argumentos de salida. *)
    id_out : seq arg_desc;
    (* Signatura de la instruccion. *)
    id_args_kinds : seq arg_position_desc;
  }.
```

Figura 4.1: Descripción de una instrucción *assembly*.

argumento o resultado, como si es implícito o explícito, si tiene alguna restricción (por ejemplo que no puede ser `RSP`), su posición en la sintaxis que se use, etcétera. La signatura es una disyunción de las posibles formas que pueden tomar los argumentos de la instrucción, llamadas *argument kinds*. Por ejemplo, en x86 la instrucción `MOV` toma como primer argumento un registro o posición de memoria y como segundo un registro, inmediato o posición de memoria, por lo que su signatura es

```
[:: [:: AKReg; AKMem ]; [:: AKReg; AKImm; AKMem ] ]
```

Las descripciones de instrucción tiene más datos, utilizados por ejemplo para el *pretty-printing*, y además invariantes sobre la descripción (como que las listas `id_tin` e `id_in` tienen el mismo largo) que no se desarrollan en este trabajo.

Una particularidad de ARMv7 es que varias instrucciones tienen tres variantes: pueden modificar las *flags* (indicado por el sufijo `S` en la instrucción, por ejemplo `ADDS r0, r1, r2`), hacer un *shift* a uno de sus operandos que sea un registro (por ejemplo `ADD r0, r1, r2 LSL 3`), y ejecutarse condicionalmente (indicado por el sufijo correspondiente a la condición, por ejemplo `ADDeq`, `ADDlt` o `ADDge`). Se buscó entonces definir las instrucciones de manera que estas transformaciones se puedan hacer de manera uniforme, definiendo funciones que convierten una descripción de instrucción en una que modifica las *flags*, o que se ejecuta condicionalmente. Una dificultad de esta tarea es que el tipo de `id_semi` depende de los valores de `id_tin` e `id_tout`, por lo que se deben emplear ciertos trucos para convencer al chequeador de tipos de Coq de que el término está bien formado. El la Figura 4.2 se muestra la descripción de la instrucción `ADD`.

4.2.2. Lowering

Como se describió en la Sección 3.2, el propósito de esta fase es reemplazar las asignaciones y las condiciones por operaciones *assembly*. Esta fase es específica de

```
Definition id_ADD (opts : arm_options) : instr_desc :=
  let x :=
    {|
      id_tin := [:: word U32; word U32 ];
      id_tout := [:: word U32 ];
      id_semi := fun wn wm => Some (wn + wm);
      id_in := [:: ADExplicit 1; ADExplicit 2 ];
      id_out := [:: ADExplicit 0 ];
      id_args_kinds :=
        [:: [:: AKReg ]; [:: AKReg ]; [:: AKReg; AKImm ] ];
    |}
  in
  let x := if has_shift opts is Some sk then mk_shift sk x else x in
  let x := if set_flags opts then mk_set_flags x else x in
  let x := if is_conditional opts then mk_conditional x else x in
  x.
```

Figura 4.2: Descripción de la instrucción `ADD`.

```
Definition get_shift (e : expr) : option (var * shift_kind * int) :=
  if e is Eapp2 op (Evar v) (Econst z)
  then
    if shift_of_sop2 op is Some sh
    then Some (v, sh, n)
    else None
  else
    None.
```

Figura 4.3: *Pattern matching* para *shifts* opcionales.

cada arquitectura, puesto que este reemplazo depende del set de instrucciones de la arquitectura. Básicamente, la fase consiste en hacer *pattern matching* en ciertas expresiones (las asignadas en las asignaciones, las condiciones en las guardas) buscando patrones que coincidan con una instrucción de la arquitectura. Se reemplazan instrucciones de la forma x `= y + z;` por x `= #ADD(y, z);` por ejemplo.

En el caso específico de ARM, se debe tener en cuenta que las instrucciones pueden tener un *shift* en uno de sus operandos, y entonces se debe compilar x `= y & z << 3` en x `= #AND_LSL(y, z, 3)`. Para esto se usa la función de la Figura 4.3, que examina la estructura de una expresión para determinar si es un *shift* válido de un entero sobre una variable.

La prueba de corrección de esta fase se hace por inducción en las instrucciones. Para el caso de la asignación se muestra que

$$[\![\mathsf{s}, \mathsf{lv}\ \texttt{=(ty)}\ \mathsf{e}\texttt{;}]\!]_{\mathsf{i}} \Downarrow \mathsf{s}' \implies [\![\mathsf{s}, \mathsf{lvs}\ \texttt{=}\ \texttt{\#op(es);}]\!]_{\mathsf{i}} \Downarrow \mathsf{s}'$$

donde lvs, op, y es son el resultado de la transformación de la asignación, y $[\![\_, \_]\!]_{\mathsf{i}} \Downarrow \_$ es el juicio definido en 3.2.1. La semántica definida en la descripción de la instrucciones cobra gran importancia en la prueba de esta propiedad.

Para las guardas se prueba una propiedad análoga, pero permitiendo que los estados difieran en un conjunto fijo de variables, que es un parámetro de fase, y representan las *flags* de la arquitectura. Los estados pueden diferir en el valor de las *flags* ya que es en ellas que se almacena el resultado de evaluar la condición.

Cambios en el código del compilador. La generalización del compilador implicó modificar 65 archivos, agregando 4300 líneas de código y eliminando 3000. Se crearon archivos nuevos para los parámetros de fase y de arquitectura. La mayoría de las líneas eliminadas fueron reemplazadas por una similar en la que se usa una de las generalizaciones introducidas.

La instanciación de ARM Cortex M4 implicó modificar 43 archivos, agregando 6700 líneas de código nuevas y eliminando 300. La mayor parte de las líneas agregadas corresponden a la declaración de las instrucciones y a la prueba de corrección de la fase Lowering (archivos `arm_instr_decl.v` y `arm_lowering_proof.v`). Se agregaron 32 instrucciones, con sus variantes condicional (se implementan las 14 condiciones de la arquitectura), escribiendo *flags* y con *shift* (se implementan 4 de los 5 shifts existentes). Además, se agregaron 40 archivos (un total de 3500 líneas) de casos de *test*, entre los cuales se encuentran: dos archivos por instrucción (uno para el caso *inline assembly* y otro para el de Lowering, en todas las variantes de la instrucción) para la mayoría de las instrucciones, uno para el Lowering de expresiones booleanas a combinaciones de *flags*, y una implementación de la permutación criptográfica Gimli [13].

5 Conclusión

En este trabajo se estudiaron las partes esenciales del compilador de Jasmin y se describió una extensión para agregar soporte a nuevas arquitecturas, en particular ARM Cortex M4. Las tareas involucradas en este proyecto no fueron perfectamente claras desde un principio: inicialmente, busqué agregar soporte para la arquitectura RISC-V al compilador, para lo que necesité generalizar distintas fases como se desarrolló en las Subsecciones 4.1.1 y 4.1.2. Luego de asegurar que las generalizaciones eran prometedoras, abstraje estas generalizaciones con los patrones de modelo de arquitectura, parámetros de arquitectura y parámetros de fase como se expuso en la Subsección 4.1.3. A la hora de instanciar la arquitectura se decidió que soportar ARM Cortex M4 sería más útil que RISC-V, por lo que debí estudiar en mayor detalle esta arquitectura y el modelo de arquitectura del compilador, instanciarla, definir sus instrucciones e implementar la fase de Lowering (ver la Subsección 4.2.1).

El resultado de estas extensiones es que el compilador se relaciona con la arquitectura a la que se compila de manera abstracta. Agregar soporte a nuevas arquitecturas ahora supone simplemente instanciar el modelo, definir sus instruciones, e implementar la fase de Lowering; es difícil imaginar un compilador para el que esta tarea implique menos requerimientos. La abstracción también influye la usabilidad del compilador: normalmente un compilador se compila para una arquitectura en específico, y es usual que haya inconvenientes compilando un compilador para una arquitectura diferente de la en la que se lo compila (por ejemplo, compilar GCC para ARM en una computadora x86-64). Esto no es el caso para el compilador de Jasmin: un solo binario puede compilar código para cualquier arquitectura instanciada.

Proveer soporte para varias arquitecturas es importante para promover el uso de Jasmin y hacerlo, así como a los demás proyectos que constituyen Formosa, una herramienta más establecida. De esta manera se aspira a promover esquemas y librerías criptográficas de mayor calidad: un esquema formalizado, verificado y que cuenta con una una implementación eficiente, segura y correcta tiene más chances de ser estadarizado, usado en desarollos de importancia y generar un impacto positivo. Si bien Jasmin es un lenguaje de muy bajo nivel, característica que se discutió en la Sección 3.1, sus diferencias con *assembly* alivian la tarea de desarrollo y verificación considerablemente. La sintaxis al estilo de C, las variables y el flujo de control estructurado hacen que desarrollar programas sea más sencillo y que razonar sobre ellos sea posible, tanto mediante *tests*, análisis estáticos o pruebas formales. Si bien un programa Jasmin en principio no puede compilarse a distintas arquitecturas, como es el caso en lenguajes de alto nivel, adaptar un programa a otra arquitectura es mucho más sencillo que en *assembly*, y luego se puede verificar rigurosamente que el comportamiento del nuevo programa es equivalente al original.

Los fundamentos teóricos y herramientas que se usaron para el estudio y desarrollo descriptos en este trabajo, presentadas en los Capítulos 2 y 3, probaron ser imprescindibles. La, quizás elemental, teoría de lenguajes de programación expuesta en el Capítulo 2 jugó un rol muy importante para comprender el funcionamiento del compilador. En mi experiencia, fue clave abordar el estudio del compilador entendiéndolo como una transformación entre sus lenguajes internos, y luego pensar dichos lenguajes en términos de sintaxis, invariantes (o juicios de tipado) y semántica. Este análisis de las fases como transformaciones entre lenguajes me posibilitó ver la corrección de cada fase de compilación como un teorema de preservación de semánticas. También, encontré que Coq, como lenguaje y asistente de pruebas, resultó muy apropiado y flexible para la tarea de implementar, modelar y probar el compilador; siendo las definiciones y enunciados de teoremas agradablemente similares a sus contrapartes teóricos.

Por otro lado, es innegable el compilador de Jasmin es considerablemente complejo a pesar de que provee apenas una fracción de la funcionalidad de otros compiladores como GCC [30] o Clang [20]. Esta complejidad no sólo se refleja en los teoremas y las pruebas de corrección: los patrones de programación usados en la implementación del compilador se ven afectados por la necesidad de verificarlos y las características del entorno (es decir, Coq). Esto afecta cuestiones simplemente estilísticas (como usar proyecciones en lugar de *pattern matching* para tuplas) pero también el diseño de algunos algoritmos, por ejemplo, para los que se separa completamente la implementación de su prueba de corrección. El diseño del compilador mismo se ve influenciado por Coq y su entorno: el ejemplo más grave de esto es el modelado de la arquitectura y sus instanciaciones, problemas que han sido estudiados en un gran número de trabajos previos [29, 5, 47], pero cuya inclusión hubiera significado un esfuerzo bastante mayor a reimplementarlos de cero y a la medida.

Trabajo futuro. De ambas partes de este trabajo surgen inmediatamente posibles trabajos futuros: de la generalización del compilador, agregar nuevas abstracciones, extender las existentes, e instanciar nuevas arquitecturas; y de la instanciación de ARM, extender este modelo para abarcar más instrucciones y comportamientos que se omitieron por simplicidad.

El compilador se beneficiaría de abstracciones nuevas para incrementar la cantidad de código compartido entre las arquitecturas, principalmente en la fase Lowering. Específicamente, la detección de expresiones booleanas y su traducción a *flags* podrían ser independientes de la arquitectura utilizando una abstracción que introduje generalizando la fase Propagate inline (que no se desarrolló en este trabajo). La abstracción consiste en un lenguaje para combinaciones de *flags* con una semántica abstracta, que instanciada en expresiones booleanas permitiría compartir un cuarto del código de la implementación de Lowering. Por otro lado, podrían agregarse al modelo de arquitectura una noción primitiva de instrucciones vectorizadas y de punto flotante, para permitir un razonamiento más granular en estos casos.

En la instanciación de ARM solo se declaran las instrucciones necesarias para para

testear la propuesta e implementar una permutación criptográfica sencilla, Gimli [13]. Además, la extracción de definiciones EasyCrypt aún necesita algunos retoques para ser completamente funcional en la nueva arquitectura. Actualmente se está discutiendo agregar una nueva arquitectura, OpenTitan [43], un microprocesador abierto enfocado en la seguridad, que no cuenta con soporte en ningún compilador público. Como este chip aún está en desarrollo, se espera que los procesos de modelarlo y de implementar rutinas criptográficas para esta arquitectura en Jasmin informen el diseño del set de instrucciones.

Bibliografía

[1] José Bacelar Almeida y col. «Jasmin: High-Assurance and High-Speed Cryptography». En: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Association for Computing Machinery, oct. de 2017, págs. 1807-1823. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134078.

[2] Ross J. Anderson. «Why cryptosystems fail». En: *Communications of the ACM* 37.11 (nov. de 1994), págs. 32-40. ISSN: 0001-0782. DOI: 10.1145/188280.188291.

[3] Arm Limited. *Arm Cortex-M4 Processor Technical Reference Manual*. Ver. r0p1. 2020. URL: https://documentation-service.arm.com/static/5fce431be167456a35b36ade.

[4] Arm Limited. *Armv7-M Architecture Reference Manual*. Ver. E.e. 2021. URL: https://documentation-service.arm.com/static/5f8ff0a1f86e16515cdbf82c.

[5] Alasdair Armstrong y col. «ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS». En: *Proceedings of the ACM on Programming Languages* 3.POPL (ene. de 2019), 71:1-71:31. DOI: 10.1145/3290384.

[6] James Ball, Julian Borger y Glenn Greenwald. *Revealed: how US and UK spy agencies defeat internet privacy and security*. The Guardian. Sep. de 2013. URL: https://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security.

[7] Henk Pieter Barendregt. *The Lambda Calculus*. Vol. 103. Studies in Logic and the Foundations of Mathematics. Elsevier, 1984. ISBN: 9780080933757. URL: https://www.elsevier.com/books/the-lambda-calculus/barendregt/978-0-444-87508-2.

[8] Gilles Barthe y col. «Computer-Aided Security Proofs for the Working Cryptographer». En: *Advances in Cryptology – CRYPTO 2011*. Ed. por Phillip Rogaway. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, págs. 71-90. ISBN: 978-3-642-22792-9. DOI: 10.1007/978-3-642-22792-9_5.

[9] Gilles Barthe y col. «EasyCrypt: A Tutorial». En: *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*. Ed. por Alessandro Aldini, Javier Lopez y Fabio Martinelli. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, págs. 146-166. ISBN: 978-3-319-10082-1. DOI: 10.1007/978-3-319-10082-1_6. URL: https://doi.org/10.1007/978-3-319-10082-1_6.

[10] Mihir Bellare y Phillip Rogaway. *Introduction to Modern Cryptography*. Class notes. Mayo de 2005.

[11] Daniel J. Bernstein. «Cryptographic competitions». En: *Cryptology ePrint Archive, Paper 2020/1608* (2020). URL: https://eprint.iacr.org/2020/1608.

[12] Daniel J. Bernstein, Tanja Lange y Ruben Niederhagen. «Dual EC: A Standardized Back Door». En: *LNCS Essays on The New Codebreakers - Volume 9100*. Berlin, Heidelberg: Springer-Verlag, nov. de 2015, págs. 256-281. ISBN: 978-3-662-49300-7. DOI: 10.1007/978-3-662-49301-4_17. URL: https://doi.org/10.1007/978-3-662-49301-4_17.

[13] Daniel J. Bernstein y col. «Gimli: A Cross-Platform Permutation». En: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. por Wieland Fischer y Naofumi Homma. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, págs. 299-320. ISBN: 978-3-319-66787-4. DOI: 10.1007/978-3-319-66787-4_15.

[14] Dan Boneh y Victor Shoup. «A Graduate Course in Applied Cryptography». Draft version 0.5. 2020. URL: https://toc.cryptobook.us/.

[15] Billy B. Brumley y col. «Practical Realisation and Elimination of an ECC-Related Software Bug Attack». En: *Topics in Cryptology – CT-RSA 2012*. Ed. por Orr Dunkelman. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, págs. 171-186. ISBN: 978-3-642-27954-6. DOI: 10.1007/978-3-642-27954-6_11.

[16] Sunjay Cauligi y col. «SoK: Practical Foundations for Software Spectre Defenses». En: *IEEE Symposium on Security and Privacy (S&P)*. IEEE. Mayo de 2022. DOI: 10.48550/arXiv.2105.05801.

[17] NIST Computer Security Resource Center. *Cryptographic Algorithm Validation Program*. NIST. URL: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/validation-search.

[18] Alonzo Church. «A formulation of the simple theory of types». En: *The Journal of Symbolic Logic* 5.2 (jun. de 1940), págs. 56-68. ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2266170.

[19] Alonzo Church. «A Set of Postulates for the Foundation of Logic». En: *Annals of Mathematics* 33.2 (1932), págs. 346-366. ISSN: 0003-486X. DOI: 10.2307/1968337.

[20] *Clang*. URL: https://clang.llvm.org/.

[21] Thierry Coquand. «Une théorie des constructions». These de doctorat. Paris 7, ene. de 1985. URL: https://www.theses.fr/1985PA07F126.

[22] Thierry Coquand y Gérard Huet. «The calculus of constructions». En: *Information and Computation* 76.2 (feb. de 1988), págs. 95-120. ISSN: 0890-5401. DOI: 10.1016/0890-5401(88)90005-3.

[23] Thierry Coquand y Christine Paulin-Mohring. «Inductively defined types». En: *COLOG-88*. Ed. por Per Martin-Löf y Grigori Mints. Lecture Notes in Computer Science. Springer, 1990, págs. 50-66. ISBN: 978-3-540-46963-6. DOI: 10.1007/3-540-52335-9_47.

[24] Ian Cutress. *New #1 Supercomputer: Fugaku in Japan, with A64FX, take Arm to the Top with 415 PetaFLOPs*. AnandTech. Jun. de 2020. URL: https://www.anandtech.com/show/15869/new-1-supercomputer-fujitsus-fugaku-and-a64fx-take-arm-to-the-top-with-415-petaflops.

[25] *CVE Program*. URL: https://www.cve.org.

[26] Michael Dummet. *The Logical Basis of Metaphysics*. Harvard University Press, 1976. Cap. Stability. ISBN: 0-674-53786-6.

[27] The Editors of Encyclopaedia Britannica. *Computer security*. En: *Encyclopædia Britannica*. 2022.

[28] *Formosa*. URL: https://formosa-crypto.org/.

[29] Anthony Fox y Magnus O. Myreen. «A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture». En: *Interactive Theorem Proving*. Ed. por Matt Kaufmann y Lawrence C. Paulson. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, págs. 243-258. ISBN: 978-3-642-14052-5. DOI: 10.1007/978-3-642-14052-5_18.

[30] *GCC, the GNU Compiler Collection*. URL: https://gcc.gnu.org/.

[31] Gerhard Gentzen. «Investigations into Logical Deduction». Trad. por Manfred E. Szabo. En: *American Philosophical Quarterly* 1.4 (1964), págs. 288-306. ISSN: 0003-0481. URL: https://www.jstor.org/stable/20009142.

[32] Herman Geuvers. «Induction Is Not Derivable in Second Order Dependent Type Theory». En: *Typed Lambda Calculi and Applications*. Ed. por Samson Abramsky. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, págs. 166-181. ISBN: 978-3-540-45413-7. DOI: 10.1007/3-540-45413-6_16.

[33] Jean-Yves Girard. «Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur». Tesis doct. Université de Paris, 1972.

[34] SoftBank Group. *SoftBank Group Annual Report of 2020*. https://group.softbank/system/files/pdf/ir/financials/annual_reports/annual-report_fy2020_01_en.pdf. 2020.

[35] S. C. Kleene y J. B. Rosser. «The Inconsistency of Certain Formal Logics». En: *Annals of Mathematics* 36.3 (1935), págs. 630-636. ISSN: 0003-486X. DOI: 10.2307/1968646.

[36] Jeff Larson. *Revealed: The NSA's Secret Campaign to Crack, Undermine Internet Security*. ProPublica. Sep. de 2013. URL: https://www.propublica.org/article/the-nsas-secret-campaign-to-crack-undermine-internet-encryption.

[37] David Lazar y col. «Why does cryptographic software fail? a case study and open problems». En: *Proceedings of 5th Asia-Pacific Workshop on Systems*. APSys '14. New York, NY, USA: Association for Computing Machinery, jun. de 2014, págs. 1-7. ISBN: 978-1-4503-3024-4. DOI: 10.1145/2637166.2637237. URL: https://doi.org/10.1145/2637166.2637237.

[38] Pierre Letouzey. «A New Extraction for Coq». En: *Types for Proofs and Programs*. Ed. por Herman Geuvers y Freek Wiedijk. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, págs. 200-219. ISBN: 978-3-540-39185-2. DOI: 10.1007/3-540-39185-1_12.

[39] Pierre Letouzey. «Extraction in Coq: An Overview». En: *Logic and Theory of Algorithms*. Ed. por Arnold Beckmann, Costas Dimitracopoulos y Benedikt Löwe. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, págs. 359-369. ISBN: 978-3-540-69407-6. DOI: 10.1007/978-3-540-69407-6_39.

[40] *LibJade*. URL: https://github.com/formosa-crypto/libjade.

[41] *Max Planck Institute for Security and Privacy (MPI-SP)*. URL: https://www.mpi-sp.org.

[42] Isaiah Mayersen. «China's Alibaba is making a 16-core, 2.5 GHz RISC-V processor». En: *Techspot* (28 de jul. de 2019). URL: https://www.techspot.com/news/81177-china-alibaba-making-16-core-25-ghz-risc.html (visitado 30-08-2021).

[43] *OpenTitan*. URL: https://opentitan.org/.

[44] Christine Paulin-Mohring. «Extracting F omega programs from proofs in the calculus of constructions». En: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '89. Association for Computing Machinery, ene. de 1989, págs. 89-104. ISBN: 978-0-89791-294-5. DOI: 10.1145/75277.75285. URL: https://doi.org/10.1145/75277.75285.

[45] Nicole Perlroth, Jeff Larson y Scott Shane. *N.S.A. Able to Foil Basic Safeguards of Privacy on Web*. The New York Times. Sep. de 2013. URL: https://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html.

[46] Calista Redmond. «How the European Processor Initiative is Leveraging RISC-V for the Future of Supercomputing». En: *RISC-V Community News* (20 de ago. de 2019). URL: https://riscv.org/news/2019/08/how-the-european-processor-initiative-is-leveraging-risc-v-for-the-future-of-supercomputing/ (visitado 30-08-2021).

[47] *RISCV Sail Model*. URL: https://github.com/riscv/sail-riscv/.

[48] Ronnald L. Rivest. *Handbook of theoretical computer science (vol. A): algorithms and complexity*. Ed. por Jan van Leeuwen. Cambridge, MA, USA: MIT Press, 1991. Cap. Cryptography, págs. 718-755. ISBN: 978-0-444-88071-0.

[49] Daniel Schatz, Rabih Bashroush y Julie Wall. «Towards a More Representative Definition of Cyber Security». En: *Journal of Digital Forensics, Security and Law* 12.2 (jun. de 2017). ISSN: (Print) 1558-7215. DOI: https://doi.org/10.15394/jdfsl.2017.1476. URL: https://commons.erau.edu/jdfsl/vol12/iss2/8.

[50] Bruce Schneier. *NSA Surveillance: a Guide to Staying Secure*. The Guardian. Sep. de 2013. URL: https://www.schneier.com/essays/archives/2013/09/nsa_surveillance_a_g.html.

[51] C. E. Shannon. «A mathematical theory of communication». En: *The Bell System Technical Journal* 27.3 (jul. de 1948), págs. 379-423. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[52] C. E. Shannon. «Communication theory of secrecy systems». En: *The Bell System Technical Journal* 28.4 (oct. de 1949), págs. 656-715. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1949.tb00928.x.

[53] The Coq Development Team. «The Coq Proof Assistant». En: *Zenodo* (ene. de 2021). DOI: 10.5281/zenodo.4501022.

[54] Texas Instruments. *AM572x Sitara Processors Silicon Revision 2.0*. 2019. URL: https://www.ti.com/lit/gpn/am5728.

[55] The Agda Team. *Agda 2*. URL: https://github.com/agda/agda.

[56] Philip Wadler. «The Girard–Reynolds isomorphism (second edition)». En: *Theoretical Computer Science*. Festschrift for John C. Reynolds's 70th birthday 375.1 (mayo de 2007), págs. 201-226. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.12.042.