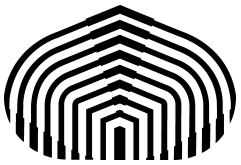


Universidad Simón Bolívar
Decanato de Estudios Profesionales
Coordinación de Ingeniería de la Computación

Formalización de un lenguaje imperativo con arreglos y apuntadores en Isabelle/HOL

Por:
Gabriela Limonta
Realizado con la asesoría de:
Federico Flaviani

PROYECTO DE GRADO
Presentado ante la Ilustre Universidad Simón Bolívar
como requisito parcial para optar al título de
Ingeniero de Computación

Sartenejas, noviembre de 2015

Resumen

C es un lenguaje de programación ampliamente utilizado, el cual es particularmente popular en la implementación de sistemas de operación y aplicaciones de sistemas embebidos. Es frecuentemente utilizado en el desarrollo de compiladores, librerias e interpretadores para otros lenguajes de programación debido a su eficiencia. La eficiencia de C se debe al hecho de que está mas cercano a la máquina que otros lenguajes de alto nivel y presenta un bajo *overhead*. Desafortunadamente, parte de la semántica para el lenguaje se describe utiliando el idioma inglés, lo cual hace que esta definición sea propensa a ambigüedades.

Este trabajo tiene tres objetivos principales. En primer lugar, formalizar la semántica de un lenguaje imperativo (con arreglos y apuntadores) que represente un subconjunto determinístico del lenguaje C y escribir un interpretador para el mismo. Segundo, generar código C a partir de la semántica que sea compilable y ejecutable. Por último, crear una *suite* de pruebas y un arnés de pruebas que permitan verificar que el comportamiento del programa generado en C coincide con el comportamiento de la semántica formalizada.

Una semántica de pasos cortos se formaliza para el lenguaje en el demostrador de teoremas Isabelle/HOL. Luego, un interpretador para el lenguaje dentro del ambiente de Isabelle/HOL se escribe, lo cual permite la ejecución de programas escritos en la semántica. Además, se demuestra que este interpretador es correcto con respecto a la semántica formal. Con el fin de traducir programas de la semántica formal a código C, se escribe un generador de código. Este generador de código uiliza el proceso de *pretty printing* para traducir la semántica formal a representaciones en *string* que pueden ser luego exportadas a un archivo. Para poder verificar que el proceso de traducción se hace de manera adecuada, el estado final del programa generado se compara al estado final esperado para determinar si son equivalentes. Por lo tanto, se define un método para generar instrucciones de prueba que verifican la condición de equivalencia para cualquier programa que se traduce a C. Un arnés de pruebas se presenta en el ambiente de Isabelle/HOL que traduce las instrucciones de pruebas a macros en C que se definen fuera de este ambiente. Finalmente, una *suite* de pruebas se presenta junto con programas de ejemplo en el lenguaje.

Palabras clave: semántica, lenguajes, métodos formales.

Agradecimientos

Me gustaría expresar mi gratitud a mi tutor Federico Flaviani por su ayuda con este proyecto y su gran apoyo, sobretodo en las fases finales.

Me gustaría agradecer a mi supervisor, Prof. Tobias Nipkow, cuyo conocimiento y pasión en esta área de investigación ha sido una fuente de inspiración y motivación desde el inicio de mi año de intercambio en la Technische Universität München. También quisiera agradecerle por la brindar la oportunidad de trabajar en esta interesante área de investigación.

Además, me gustaría dar un agradecimiento especial a mi asesor, Dr. Peter Lammich, por su orientación y apoyo constante durante este proyecto. Él me guió a lo largo de este proyecto y me dió consejos y conocimientos útiles.

Por otra parte, deseo ofrecer mi mas sincero agradecimiento a Antti Sonkeri y Carlos Pérez por su valiosa ayuda al leer los primeros borradores de este trabajo.

También me gustaría agradecer de todo corazón a John Delgado y a Manuel Gómez por su amistad y apoyo constante a lo largo de los últimos años.

Por último, pero no menos importante, quiero expresar mi más profunda gratitud y amor a mi familia. Les debo todo, ellos son mi inspiración y siempre me han motivado a perseverar aun cuando los tiempos se ponen difíciles. Eugenio, Gaudy, Santiago y Juana, sin ustedes esto no hubiera sido posible, gracias.

Índice general

Índice de figuras

Índice de Tablas

Dedicatoria

A todos mis familiares y amigos que ayudaron a hacer posible este trabajo.

Capítulo 1

Introducción

En este capítulo se discute la motivación para realizar este trabajo. Seguido de una breve descripción de los conceptos relacionados a semántica e Isabelle/HOL.

Luego, se describen las características del lenguaje utilizado en este trabajo.

Finalmente, se describe el contenido del resto del trabajo.

1.1. Motivación

El objetivo de este trabajo es formalizar al semántica de un lenguaje imperativo (incluyendo apuntadores y arreglos) que representa un subconjunto del lenguaje C y luego, generar código ejecutable del mismo.

C es un lenguaje ampliamente utilizado. Es especialmente popular en la implementación de sistemas operativos y aplicaciones de sistemas embebidos. Dado que C es mas cercano a la máquina en comparación con otros lenguajes de alto nivel y presenta bajo *overhead*, permite la implementación eficiente de algoritmos y estructuras de datos. Debido a su eficiencia, a menudo es usado en el desarrollo de compiladores, librerías e interpretadores para otros lenguajes de programación.

Desafortunadamente, parte de la semántica para el lenguaje C descrita en el estándar (ISO/IEC, 2007) es propensa a ambigüedades debido al uso del idioma inglés para describir el comportamiento de un programa. El uso de constructores matemáticos formales eliminaría estas ambigüedades, aunque la formalización de la semántica para la totalidad del lenguaje C no es una tarea fácil y de hecho es una que ha sido objeto de mucha investigación en el área de la semántica.

A pesar de que la semántica definida en el presente trabajo cubre un subconjunto limitado del lenguaje C, es lo suficientemente expresiva como para permitir la implementación de algoritmos tales como algoritmos de ordenamiento, búsqueda en arboles, etc.

Otro de los objetivos de este trabajo es hacer que la semántica formal sea ejecutable en el ambiente de Isabelle/HOL. La semántica formalizada es determinística, lo que permite la definición de un interpretador que pueda, efectivamente, ejecutar la semántica. Este interpretador retornará el estado final resultante de la ejecución de la semántica.

La generación de código C que pueda ser ejecutable está entre los objetivos planteados en este trabajo. La semántica formal definida en este trabajo corresponde a la semántica del lenguaje C implementada por un compilador. Se proporciona un mecanismo mediante el cual se pueden traducir los programas escritos en la semántica formal a código C que puede ser compilado y ejecutado en una máquina. El código generado, al ser compilado y ejecutado en una máquina, presentará el mismo comportamiento que el programa interpretado dentro del ambiente de Isabelle/HOL. Esto permite la implementación de algunos algoritmos verificados utilizando la semántica y la generación de código C eficiente a partir de la misma que puede ser compilado y ejecutado en la máquina.

Finalmente, también es objetivo de este trabajo verificar que la semántica sea compatible con un compilador de C real. Para ello se define un arnés de pruebas y una *suite* de pruebas que tienen el propósito de aumentar la confianza en el proceso de traducción, es decir, la semántica del programa no es cambiada por el proceso de traducción a lenguaje C. El proceso de pruebas intenta comprobar que el estado final de un programa ejecutado en el ambiente de Isabelle/HOL y el estado final del programa generado, que es compilado y ejecutado fuera de este ambiente, serán iguales (excepto para el caso en el que se presente una falla al intentar asignar memoria dinámica). Para garantizar esto, se escribe una librería para el arnés de pruebas en C que se utiliza para realizar pruebas generadas automáticamente (para los programas escritos en la semántica) que se encargan de comparar el estado final de la semántica ejecutada en el ambiente Isabelle/HOL con aquel del programa compilado.

1.2. Marco Teórico

1.2.1. Semántica de un lenguaje de programación

En esta sección se da una breve introducción a los conceptos relacionados a semántica con los que el lector debe estar familiarizado para leer el contenido de este trabajo.

Definición

La semántica de un lenguaje de programación es el significado de programas en ese lenguaje. Según Tennent (1991), para poder definir y respaldar el significado de un programa en un lenguaje de programación, se necesita una teoría matemática de la semántica de los lenguajes de programación que sea rigurosa.

Como es señalado por Nielson y Nielson (2007), el carácter riguroso de este estudio se debe al hecho de que puede revelar ambigüedades o complejidades subyacentes en los documentos definidos en lenguaje natural, y también que este rigor matemático es necesario para pruebas de correctitud. Para muchos lenguajes de programación grandes, por ejemplo el lenguaje C, su documento de referencia (donde la semántica del lenguaje se explica) se encuentra escrito en lenguaje natural. Dada la ambigüedad presente en el lenguaje natural, esto puede llevar a dificultades cuando se intenta razonar sobre los programas escritos en esos lenguajes de programación.

La falta de una semántica definida matemáticamente de una forma rigurosa hace que sea difícil para los desarrolladores escribir herramientas precisas y correctas para el lenguaje. Debido a las ambigüedades, parte del comportamiento del lenguaje está sujeto a la interpretación del lector. Mediante el uso de términos definidos matemáticamente podemos eliminar esta posible ambigüedad. Si cada término se describe matemáticamente, entonces podemos asegurar que el significado definido en la semántica de un lenguaje solo puede ser uno y no puede tener diferentes interpretaciones.

Con el fin de aclarar la definición y los diferentes tipos de semántica, se presenta un ejemplo considerado relevante de Nielson y Nielson (2007). Se toma el siguiente programa

$$z := x; x := y; y := z$$

donde ":=" es una asignación a una variable y ";" es la secuenciación de instrucciones. Sintácticamente este programa está compuesto por tres instrucciones separadas por ";", donde cada instrucción está compuesta por una variable, el símbolo ":=" y una segunda variable.

La *semántica* de este programa es el significado del mismo. Semánticamente, este programa intercambia los valores de x e y (usando z como una variable temporal).

Tipos de semántica

En la sección anterior, se presento un programa ejemplo y una explicación aproximada de su significado en lenguaje natural. Esta explicación se podría haber hecho con mayor claridad y rigor al explicar formalmente el significado de las instrucciones, especialmente el significado de las instrucciones de asignación y secuenciación.

Existen muchos enfoques diferentes sobre cómo formalizar la semántica de un lenguaje de programación, dependiendo de la finalidad. A continuación se presentan los enfoques más utilizados:

Semántica operacional

Una semántica se define utilizando el enfoque operacional cuando el foco se pone en *cómo* se ejecuta un programa. Se puede considerar como una abstracción de la ejecución del programa en una máquina (Nielson and Nielson, 2007). Dado un programa, su explicación operacional representa cómo se ejecuta el mismo dado un estado inicial.

Tomando el ejemplo dado anteriormente, dar una interpretación de semántica operacional para ese programa se reduce a definir cómo las instrucciones de asignación y secuenciación se ejecutan. En un primer enfoque intuitivo se pueden distinguir dos reglas básicas:

- Para ejecutar una secuencia de instrucciones, cada instrucción se ejecuta en un orden de izquierda a derecha.
- Para ejecutar una instrucción de asignación entre dos variables, el valor de la variable del lado derecho se determina y se asigna a la variable del lado izquierdo.

Existen dos tipos diferentes de semánticas operacionales: *semántica de pasos cortos* (o semántica operacional estructurada) y *semántica de pasos largos* (o semántica natural). Se procederá a introducir ambos conceptos y a construir una interpretación para el ejemplo dado anteriormente haciendo uso de ambas semánticas.

Semántica de pasos largos Este tipo de semántica representa la ejecución de un programa desde un estado inicial hasta un estado final en un solo paso, por lo tanto, no permite la inspección explícita de estados de ejecución intermedios (Nipkow and Klein, 2014).

Suponiendo que se tiene un estado donde la variable x tiene el valor 5, la variable y tiene el valor 7 y la variable z tiene el valor 0 y el programa del ejemplo presentado anteriormente, la ejecución del programa completo se verá de la siguiente manera:

$$\langle z := x; x := y; y := z, s_0 \rangle \to s_3$$

donde se utilizan las siguientes abreviaciones:

$$s_0 = [x \mapsto 5, y \mapsto 7, z \mapsto 0]$$
$$s_3 = [x \mapsto 7, y \mapsto 5, z \mapsto 5]$$

Sin embargo, podemos obtener la siguiente "secuencia de derivación" para el programa anterior:

$$\frac{\dfrac{\langle z := x, s_0 \rangle \to s_1 \qquad \langle x := y, s_1 \rangle \to s_2}{\langle z := x; x := y, s_0 \rangle \to s_2} \qquad \langle y := z, s_2 \rangle \to s_3}{\langle z := x; x := y; y := z, s_0 \rangle \to s_3}$$

donde se utilizan las siguientes abreviaciones:

$$s_0 = [x \mapsto 5, y \mapsto 7, z \mapsto 0]$$
$$s_1 = [x \mapsto 5, y \mapsto 7, z \mapsto 5]$$
$$s_2 = [x \mapsto 7, y \mapsto 7, z \mapsto 5]$$
$$s_3 = [x \mapsto 7, y \mapsto 5, z \mapsto 5]$$

La ejecución de $z := x$ en el estado s_0 producirá el estado s_1 y la ejecución de $x := y$ en el estado s_1 producirá el estado s_2. Por lo tanto la ejecución de $z := x; x := y$ en el estado s_0 producirá el estado s_2. Además, la ejecución de $y := z$ en el estado s_2 producirá el estado s_3. Finalmente, la ejecución de todo el programa $z := x; x := y; y := z$ en el estado s_0 producirá el estado s_3.

Semántica de pasos cortos A veces es deseable tener mayor información con respecto a los estados intermedios de un programa, es por eso que la semántica de pasos cortos existe.

Este tipo de semántica representa pequeños pasos de ejecución atómicos en un programa y permite razonar sobre qué tanto ha sido ejecutado de un programa y explícitamente inspeccionar ejecuciones parciales (Nipkow and Klein, 2014)

En este ejemplo se comienza desde el programa completo y se toman pequeños pasos (denotados por "⇒") que produce el resto del programa que queda por ejecutar luego de

ejecutar un paso corto y el estado resultante luego de ejecutar el mismo, hasta que todo el programa es ejecutado.

Suponiendo que se tiene un estado donde la variable x tiene el valor 5, la variable y tiene el valor 7 y la variable z tiene el valor 0¸ y el programa del ejemplo, se obtiene la siguiente "secuencia de derivación":

$$\begin{aligned} & \langle z := x; x := y; y := z, [x \mapsto 5, y \mapsto 7, z \mapsto 0] \rangle \\ \Rightarrow \quad & \langle x := y; y := z [x \mapsto 5, y \mapsto 7, z \mapsto 5] \rangle \\ \Rightarrow \quad & \langle y := z [x \mapsto 7, y \mapsto 7, z \mapsto 5] \rangle \\ \Rightarrow \quad & [x \mapsto 7, y \mapsto 5, z \mapsto 5] \end{aligned}$$

Lo que sucede aquí es que en en primer paso, la instrucción $z := x$ se ejecuta y el valor de la variable z cambia a 5, x e y permanecen sin cambios. Luego de la ejecución de la primera instrucción, queda el programa $x := y; y := z$. Se ejecuta la segunda instrucción $x := y$ y el valor de x cambia a 7, y y z permanecen sin cambios. Entonces, queda el programa $y := z$, después de ejecutar esta instrucción final, el valor de y cambia a 5, x y z permanecen sin cambios.

Por último, se tiene que el comportamiento de este programa es intercambiar los valores de x e y utilizando z como una variable temporal.

Semántica denotacional

La semántica denotacional deja de centrarse en *como* se ejecuta un programa y redirige su enfoque hacia el *efecto* de ejecutar el programa. Este enfoque sirve de ayuda pues da un *significado* a los programas en un lenguaje (Nipkow and Klein, 2014). Este enfoque se modela mediante el uso de funciones matemáticas. Se tiene una función por cada construcción en el lenguaje, que define su significado, y estas funciones operan sobre estados. Toman el estado inicial y produce el estado resultante de aplicar el efecto de la construcción.

Si se toma el ejemplo anterior, se pueden definir los efectos de las diferentes construcciones que tenemos: instrucciones de secuenciación y asignación.

- El efecto de una secuencia de instrucciones se define como la composición funcional de cada instrucción individual.

- El efecto de una asignación entre dos variables se define como una función que toma un estado y produce el mismo estado, donde el valor actual de la variable del lado izquierdo es actualizada con el nuevo valor de la variable del lado derecho.

Para este ejemplo en particular se obtendrían funciones de la forma $S[\![z := x]\!]$, $S[\![x := y]\!]$ and $S[\![y := z]\!]$ para cada instrucción individual. Por otra parte, para la instrucción compuesta que es el programa completo, se obtendría la siguiente función:

$$S[\![z := x; x := y; y := z]\!] = S[\![y := z]\!] \circ S[\![x := y]\!] \circ S[\![z := x]\!]$$

La ejecución del programa completo $z := x; x := y, y := z$ tendría el efecto de *aplicar* la función $S[\![z := x; x := y; y := z]\!]$ al estado inicial $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$:

$$\begin{aligned}
S[\![z := x;&x := y; y := z]\!]([x \mapsto 5, y \mapsto 7, z \mapsto 0]) \\
&= (S[\![y := z]\!] \circ S[\![x := y]\!] \circ S[\![z := x]\!])([x \mapsto 5, y \mapsto 7, z \mapsto 0]) \\
&= S[\![y := z]\!](S[\![x := y]\!](S[\![z := x]\!]))([x \mapsto 5, y \mapsto 7, z \mapsto 0]) \\
&= S[\![x := y]\!](S[\![z := x]\!])([x \mapsto 5, y \mapsto 7, z \mapsto 5]) \\
&= S[\![z := x]\!]([x \mapsto 7, y \mapsto 7, z \mapsto 5]) \\
&= [x \mapsto 7, y \mapsto 5, z \mapsto 5]
\end{aligned}$$

El enfoque está en el estado resultante que representa el efecto que el programa tuvo en un estado inicial. Es más sencillo razonar sobre los programas utilizando este enfoque ya que es similar a razonar sobre objetos matemáticos. Aunque es importante tomar en cuenta que el establecimiento de una base matemática firme para hacerlo no es una tarea trivial. El enfoque denotacional puede ser fácilmente adaptado para representar algunas propiedades de los programas. Ejemplos de esto son inicialización de variables, plegamiento de constantes (*constant folding*) y alcance de código (Nielson and Nielson, 2007).

Semántica Axiomática

También conocida como Lógica de Hoare, este enfoque final se utiliza cuando el interés recae en probar propiedades de los programas. Se puede hablar de *correctitud parcial* de un programa con respecto a una construcción, una precondición y una postcondición siempre que la siguiente implicación se cumpla:

> Si la precondición se cumple antes de que el programa se ejecute, y la ejecución del programa termina, entonces la postcondición se cumple para el estado final.

También se puede hablar de *correctitud total* de un programa con respecto a una construcción, una precondición y una postcondición siempre que la siguiente implicación se cumpla:

> Si la precondición se cumple antes de que el programa se ejecute y la ejecución del programa termina, entonces la postcondición se cumple para el estado final.

Por lo general es mas sencillo hablar sobre el concepto de correctitud parcial (Nipkow and Klein, 2014).

La siguiente propiedad de correctitud parcial se define para el programa del ejemplo:

$$\{x = n \wedge y = m\} z := x; x := y; y := z \{x = m \wedge y = n\}$$

donde $\{x = n \wedge y = m\}$ y $\{x = m \wedge y = n\}$ son la precondición y postcondición respectivamente. n y m indican los valores iniciales de x e y. El estado $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$ cumple la precondición si se toman $n = 5$ y $m = 7$. Luego de que la propiedad de correctitud parcial es *demostrada*, entonces se puede deducir que *si* el programa termina *entonces* lo hará en un estado donde x es 7 e y es 5.

Este enfoque se basa en un *sistema de demostración* o reglas de inferencia para derivar y demostrar propiedades de correctitud parcial (Nipkow and Klein, 2014). El siguiente "arbol de demostración" puede expresar una prueba de la propiedad de correctitud parcial anterior.

$$\dfrac{\dfrac{\{p_0\} z := x \{p_1\} \qquad \{p_1\} x := y \{p_2\}}{\{p_0\} z := x; x := y \{p_2\}} \qquad \{p_2\} y := z \{p_3\}}{\{p_0\} z := x; x := y; y := z \{p_3\}}$$

donde se utilizan las siguientes abreviaciones:

$$p_0 = x = n \wedge y = m$$

$$p_1 = y = m \wedge z = n$$

$$p_2 = x = m \wedge z = n$$

$$p_3 = x = m \wedge y = n$$

La ventaja de usar este enfoque es que se cuenta con una manera fácil de demostrar propiedades de un programa, dada por el sistema de demostración.

1.2.2. Isabelle/HOL

Hoy en día se cuenta con la existencia de demostradores de teoremas automatizados que ayudan en la formalización y demostración de diferentes programas. Las pruebas escritas en papel y lápiz son propensas a errores y los seres humanos son mas fáciles de engañar que a una máquina. Por lo tanto se deben aprovechar los recursos ofrecidos por una máquina para permitir que realice el trabajo pesado. Razonar sobre la semántica de un lenguaje de programación sin el uso de herramientas automatizadas se convierte en una gran tarea y, como se dijo antes, propensa a errores, por no hablar de que la certeza sobre la correctitud de las demostraciones disminuye.

Mediante el uso de un demostrador de teoremas, en el caso de este trabajo Isabelle/HOL (Nipkow et al., 2015), se puede estar seguro de que nos resultados demostrados son correctos. En el entorno de Isabelle/HOL se pueden hacer definiciones lógicas y demostrar lemas y teoremas sobre esas definiciones de una manera sólida. La semántica para el lenguaje utilizado en este trabajo está formalizada en Isabelle/HOL, así como las pruebas que acompañan a estas definiciones. Las definiciones de la semántica formal y pruebas que lo acompañan están escritas en Isabelle/HOL. No se especificarán los detalles de Isabelle/HOL en esta sección sino que se remite al lector al libro *Concrete Semantics with Isabelle/HOL* (Nipkow and Klein, 2014) en el que se muestra una introducción a Isabelle/HOL y demostración de teoremas.

Por otra parte, Isabelle/HOL también cuenta con herramientas de generación de código (Haftmann, 2015) que permiten la generación de código ejecutable en SML correspondiente a la especificación HOL de la semántica. Este código generado permitirá mas adelante la ejecución de la semántica definida. También, Isabelle/HOL permite que código escrito en Isabelle/ML esté embebido en las teorías (Wenzel, 2015), lo cual facilita el proceso de traducción a lenguaje C.

1.2.3. Chloe

En el presente trabajo se formaliza la semántica de un pequeño lenguaje de programación llamado Chloe. Este lenguaje es un subconjunto del lenguaje de programación C. Aunque la sintaxis y semántica de este lenguaje se presentan en mayor detalle en el capítulo 3 es importante mencionar que se trata de una semántica operacional de pasos cortos.

Este lenguaje tiene las siguientes características: variables, arreglos, aritmética de apuntadores, ciclos, condicionales, funciones y memoria dinámica. El alcance de este proyecto se

limitó a las características mencionadas anteriormente y hay varias características que actualmente no son soportadas por el lenguaje. Estas características son: un sistema estático de tipos que este probado que sea correcto y sólido, concurrencia, operaciones I/O, goto, etiquetas, break y continue.

Si bien será conveniente en el futuro tener las características que actualmente no cubre el alcance de este trabajo, el actual conjunto de características soportadas por Chloe son suficientes para mostrar ejemplos relevantes de programas y tiene suficiente poder expresivo como para ser Turing-completo.

1.3. Estructura del documento

El resto de este documento se divide de la siguiente manera: el capítulo 2 abarca los trabajos previos y relacionados al presente, el capítulo 3 abarca los detalles de la sintaxis y la semántica de pasos cortos definida para Chloe, el capítulo 4 abarca el proceso de traducción de un programa en la semántica de Chloe a un programa en C, a este proceso se le llama *pretty printing*, el capítulo 5 abarca el conjunto de pruebas que verifican la correctitud del proceso de traducción y finalmente, el capítulo 6 encapsula el resultado del trabajo y las conclusiones finales del mismo, así como también detalla la dirección a seguir para realizar trabajo a futuro a partir del presente.

Capítulo 2

Antecedentes y trabajos relacionados

Hay una amplia variedad de trabajos relacionados a la formalización de la semántica del lenguaje C. Limitaremos esta sección a aquellos trabajos que son directamente relevantes al nuestro. En este capítulo se procederá a presentar la lista de trabajos previos relacionados a este trabajo.

2.1. Formalización de C en HOL

El trabajo de Michael Norrish (1998) es uno muy importante y que es necesario tomar en cuenta cuando se habla de la semántica de C. En el mismo, Norrish, formaliza la semántica operacional para el subconjunto llamado Cholera del lenguaje C. Esta semántica está formalizada en el probador de teoremas HOL (Norrish and Slind, 2014).

Una de las características mas importantes de su formalización de la semántica de C es el hecho de que considera cada posible orden de evaluación para efectos de borde. Norrish prueba que expresiones *puras*[1] en su lenguaje son determinísticas.

Por otra parte, define expresiones *libres de puntos de secuencia*[2], las cuales están solapadas con el conjunto de expresiones puras pero ningún conjunto contiene al otro, y prueba que son determinísticas.

[1]Las expresiones puras están definidas como expresiones que no contienen llamadas a funciones, asignaciones, incrementos o decrementos postfijos.

[2]Las expresiones libres de puntos de secuencia están definidas como aquellas expresiones que no contienen la evaluación de alguno de los operadores lógicos &&, || o ? :, un operador coma o una llamada a función.

Norrish presenta una lógica de programación, la cual permite el razonamiento sobre programas a nivel de instrucciones. Para esto se presenta una derivación de una lógica de Hoare ara programas en C y luego se presenta un sistema para analizar los cuerpos de un ciclo y generar postcondiciones correctas a partir de ellos considerando la existencia de instrucciones `break`, `continue` y `return` en el cuerpo del ciclo.

Este trabajo es relevante al presente dado que también se formaliza la semántica de un subconjunto mas pequeño del lenguaje C. Sin embargo, el trabajo de Norrish tiene ciertas diferencias en comparación al presente trabajo. Una de ellas es que la semántica operacional definida por Norrish para las instrucciones es una semántica de pasos largos, mientras que la semántica definida en este trabajo es una semántica de pasos cortos. Por otra parte, el trabajo de Norrish se orienta a presentar la lógica de programación y al razonamiento sobre programas a nivel del probador de teoremas, mientras que este trabajo se enfoca en la generación de código y el proceso de traducción de la semántica a código ejecutable.

2.2. Modelo de memoria del compilador CompCert

El compilador CompCert es un compilador optimizador formalmente verificado que traduce código del subconjunto Clight del lenguaje de programación C (Blazy and Leroy, 2009) a código ensamblador para PowerPC. Una descripción del compilador CompCert se encuentra disponible en el capítulo 4 de Boldo et al. (2013). CompCert compila código fuente de la semántica de Clight a código ensamblador preservando la semántica del lenguaje original. Para realizar esta traducción se necesitan varios lenguajes, así como un modelo de memoria que permita el razonamiento sobre estados de memoria.

Los modelos de memoria son generalmente o demasiado concretos o demasiado abstractos. Cuando son demasiado abstractos pueden dejar de representar cosas tales como *aliasing* lo cual haría que la semántica estuviera incorrecta. Un modelo de memoria que sea demasiado concreto puede dificultar el proceso de prueba, por ejemplo, al no poder validar propiedades algebraicas que son, en efecto, válidas en el lenguaje. El modelo de memoria utilizado en el compilador CompCert (Leroy and Blazy, 2008) está en algún punto intermedio entre un modelo de bajo nivel y uno de alto nivel. La representación de la memoria tiene un conjunto de estados de memoria que están indexados por una referencia a un bloque. Cada bloque se comporta como un arreglo de bytes que puede ser indexado utilizando desplazamientos en bytes. Leroy y Blazy presentan un resumen y una descripción concreta de su modelo de memoria y tienen propiedades sobre las operaciones de memoria formalizadas y demostradas

en el asistente de pruebas Coq (Team, 2015). Una de esas propiedades es que se garantiza la separación entre dos bloques obtenidos a partir de dos llamadas diferentes a `malloc`.

La memoria de la semántica de este trabajo es modelada tomando este modelo de memoria como inspiración. El modelo de memoria de este trabajo difiere del presentado en esta sección por ser mas simple. Una de las diferencias entre ambos modelos es que el modelo de Leroy y Blazy soporta límites inferiores y superiores para acceder a bloques mientras que todos los bloques en el modelo de este trabajo son accesibles desde el índice 0 hasta la longitud del bloque. Además, cada celda de memoria en el modelo de Leroy y Blazy representa un byte, mientras que en el modelo de este trabajo cada celda de memoria contiene un valor entero o un apuntador. La idea fundamental detrás de este modelo de memoria es tomada y adaptada a las necesidades de este trabajo.

2.3. De código C a semántica

El presente trabajo tienen un enfoque de arriba a abajo (*top-down*) donde se tiene la intención de generar código en C de una especificación formal. Existe otra dirección que es relevante mencionar. El proyecto AutoCorres (Greenaway et al., 2014) reconoce código C y genera una representación monádica de alto nivel que facilita el razonamiento sobre un programa. Este trabajo permite a los usuarios razonar sobre programas en C a un nivel más alto. Se genera una especificación en Isabelle/HOL así como una demostración de correctitud en Isabelle/HOL para la traducción que se realiza. Cuenta con una abstracción del *heap* que permite el razonamiento sobre memoria para funciones *type-safe* así como una abstracción para palabras que permite que palabras de la máquina puedan ser abstraídas a números naturales y enteros, de modo que sea posible razonar acerca de los mismos a este nivel.

Es relevante considerar este enfoque de abajo a arriba (*bottom-up)* como un enfoque diferente en la verificación formal de programas. AutoCorres se utiliza en varios proyectos de verificación de C tales como la verificación de una compleja librería de grafos a gran escala, la verificación de un sistema de archivos y la verificación de un sistema operativo de tiempo real para sistemas de ala seguridad.

Capítulo 3

Sintaxis y Semántica

El lenguaje imperativo de este trabajo se llama Chloe y representa un subconjunto del lenguaje C. Un programa es una secuencia de una o más instrucciones (o comandos) escritos para llevar a cabo una tarea en una computadora. Estas instrucciones pueden contener componentes internos llamados expresiones. Una expresión es un termino que consta de valores, constantes, variables, operadores, etc. que puede ser evaluado en el contexto de un estado del programa para obtener un valor que puede ser luego utilizado en una instrucción. En las secciones siguientes se procederá a describir la sintaxis y la semántica de los programas en Chloe con más detalle.

3.1. Expresiones

3.1.1. Sintaxis

En esta sección se describe la **sintaxis abstracta** de las expresiones en el lenguaje Chloe.

En la figura 3.1 se encuentra el tipo de datos creado en Isabelle para las expresiones, donde `int` es el tipo predefinido para los enteros y `vname` significa nombre de la variable.

Se definen dos nuevos tipos de datos, uno para las expresiones y otro para las expresiones del lado izquierdo. Es importante diferenciar entre estos dos tipos en el caso en que se trabaja con expresiones que contienen apuntadores. Por ejemplo, suponiendo que se tienen las siguientes instrucciones en C:

```
foo = *bar;
*baz = 1;
```

```
type_synonym vname = string

datatype exp = Const int
             | Null
             | V      vname
             | Plus   exp exp
             | Subt   exp exp
             | Minus  exp
             | Div    exp exp
             | Mod    exp exp
             | Mult   exp exp
             | Less   exp exp
             | Not    exp
             | And    exp exp
             | Or     exp exp
             | Eq     exp exp
             | New    exp
             | Deref  exp
             | Ref    lexp
             | Index  exp exp
and
datatype lexp = Deref exp
              | Indexl exp exp
```

FIGURA 3.1: Expresiones de Chloe

donde `foo` y `bar` son variables, `1` es un valor constante, "`=`" indica una instrucción de asignación y "`*`" corresponde al operador de desreferencia.

En la primera expresión `*bar` se encuentra del lado derecho de la asignación, en este caso se quiere que `*bar` produzca un valor que pueda ser luego asignado a `foo`. Por otra parte, en la segunda expresión `*baz` se encuentra del lado izquierdo de la asignación, en este caso se quiere que `*baz` produzca una dirección a la cual se le pueda asignar el valor `1`.

Esto también ocurre con el acceso a arreglos. Para modelar correctamente la semántica de las expresiones en Chloe, es necesario contar con esta distinción entre expresiones del lado izquierdo (*l-values*) y expresiones del lado derecho (*r-values*). En las siguientes secciones al referirse a expresiones del lado derecho o *r-values* se utilizará simplemente el nombre expresiones y se utilizará expresión del LHS (por sus siglas en ingles *left-hand-side*) en lugar de *l-values* o expresiones del lado izquierdo al referirse a dichas expresiones.

Chloe soporta expresiones constantes, apuntadores a *null* y variables, así como las siguientes operaciones sobre expresiones: suma, resta, menos unario, división, módulo, multiplicación, menor que, negación, conjunción, disyunción e igualdad. También cuenta con una expresión `New` que corresponde a una llamada a `malloc` en C. Tiene operadores de desreferencia, referencia y acceso a arreglos. En C, estos son los operadores `*`, `&` y `[]`, respectivamente. Finalmente, como expresiones del LHS se tienen las operaciones de desreferencia y acceso a arreglos.

```
abbreviation INT_MIN :: int where INT_MIN ≡ - (2^(int_width - 1))
abbreviation INT_MAX :: int where INT_MAX ≡  ((2^(int_width - 1)) - 1)
```

FIGURA 3.2: Límites inferiores y superiores para los enteros

Tipos

En el lenguaje Chloe se tienen dos tipos; enteros y direcciones. Se diferencia entre los valores de tipo entero y las direcciones con el fin de definir correctamente la semántica. A continuación, se presentan los detalles de los dos tipos en el lenguaje Chloe.

Enteros Se establecen los siguientes sinónimos entre tipos en Isabelle:

```
type_synonym int_width = 64
type_synonym int_val = int_width word
```

El termino `int_width` se refiere a la precisión del valor entero. En el caso de este trabajo se supone un valor de 64 ya que se trabaja con una arquitectura de 64 bits. Este parámetro le indica a la semántica que debe suponer que se trabaja con una arquitectura de 64 bits donde los límites inferiores y superiores para un entero se definen en la figura 3.2.

Cuando se trabaja con una arquitectura diferente, este parámetro puede ser cambiado con el fin de cumplir con los requerimientos de la arquitectura.

También, los enteros se definen como palabras de longitud `int_width` (en este caso `64`). Debido a que no se utiliza el tipo predefinido `int` en Isabelle, para poder trabajar con palabras y soportar la generación de código para las mismas se utiliza la entrada *Native Word* en el *Archive of Formal Proofs* (Lochbihler, 2013).

De ahora en adelante se referirá a las palabras de longitud 64 que se utilizan para representar a los enteros en Chloe como simplemente enteros. Es importante notar que, a menos de que sea explícitamente mencionado en el texto, por simplicidad se utilizará la palabra 'entero' para referirse a las palabras de longitud 64 en lugar del tipo predefinido `int` en Isabelle.

Direcciones Se define el siguiente tipo de datos en Isabelle para representar direcciones:

```
datatype addr = nat × int
```

Una dirección es entonces un par compuesto por un número natural y un entero (el cual es de tipo `int` predefinido en Isabelle), estos representan un par `(id_bloque, desplazamiento)`. En secciones futuras se procederá a explicar el diseño de la memoria.

Valores

Los valores para una expresión se definen de la siguiente forma:

```
datatype val = NullVal | I int_val | A addr
```

donde `NullVal` corresponde a un apuntador a *null*, `I` *int_val* corresponde a un valor entero y `A` *addr* corresponde al valor de una dirección. Al evaluar una expresión se puede obtener cualquiera de estos tres valores.

Memoria

Se modela la memoria dinámica de la siguiente manera:

```
type_synonym mem = val option list option list
```

La memoria se encuentra representada como una lista de bloques asignados y cada uno de estos bloques consta de una lista de celdas con los valores en memoria. Por cada bloque hay dos posibilidades: un bloque asignado o un bloque no asignado, esto es modelado por el uso del tipo `option`, donde `Some` l (donde l es de tipo `val option list`) denota un bloque asignado y `None` uno sin asignar. Cada bloque se compone de una lista de celdas que contienen los valores en memoria. Cada celda puede tener diferentes valores dependiendo de si se encuentra sin inicializar o si posee un valor. Una celda sin sin inicializar en memoria está representada por el valor `None`. Mientras que una celda que posee un valor está representada por el valor `Some` v (donde v es de tipo `val`). Este modelo de memoria está inspirado en el trabajo de Blazy y Leroy (2008), es un modelo simplificado que fue ajustado para satisfacer las necesidades de este trabajo.

Existen cuatro operaciones principales para la gestión de memoria, estas son `new_block`, `free`, `load` y `store` y se encuentran especificadas en la figura 3.3. Cada una de estas operaciones puede fallar, por lo que su tipo de retorno es τ `option`. Los valores de ese tipo son `None` cuando la operación falla y `Some`(v) cuando es exitosa (donde v es de tipo τ).

Las funcionalidades de las operaciones de gestión de memoria se describen a continuación:

- `new_block` es la función que se encarga de asignar un nuevo bloque de memoria dinámica de un tamaño determinado. Esta función fallará en el caso donde el tamaño dado sea menor o o igual a cero, también puede fallar si un valor de un tipo diferente a entero es dado. Al ser ejecutada exitosamente, la función retorna la dirección de inicio del nuevo bloque junto con la memoria modificada.

```
new_block :: val ⇒ mem ⇒ (val × mem) option
free      :: addr ⇒ val  ⇒ visible_state ⇒ visible_state option
load      :: addr ⇒ mem ⇒ val option
store     :: addr ⇒ val  ⇒ visible_state ⇒ visible_state option
```

FIGURA 3.3: Operaciones de manejo de memoria

- `free` es la función que se encarga de liberar un bloque de memoria dinámica. Esta función fallará en el caso donde la dirección dada no sea válida en memoria. Al ser ejecutada exitosamente, la función retorna un nuevo estado que incluye la memoria actualizada.

- `load` es la función que, dada una dirección, retorna el valor almacenado en la celda de memoria denotada por la dirección dada. Esta función fallará en el caso donde la dirección dada no sea válida en memoria. Al ser ejecutada exitosamente, la función retornará el valor almacenado en memoria.

- `store` es la función que, dada una dirección y un valor, almacena dicho valor en la celda de memoria denotada por la dirección dada. Esta función fallará en el caso donde la dirección dada no sea válida en memoria. Al ser ejecutada exitosamente, la función retorna un nuevo estado que incluye la memoria actualizada.

Es importante tener en cuenta que las únicas razones por las que la asignación de memoria dinámica puede fallar en la semántica son aquellas descritas anteriormente. Debido a que se supone que la memoria es ilimitada, no existirá un caso donde una llamada a `new` falle debido a falta de memoria.

Sin embargo, como los recursos de una máquina son limitados, no se puede utilizar una cantidad ilimitada de memoria. Aquí es donde se consigue una discrepancia con lo descrito por el estándar de C. Cuando se realiza una llamada a malloc en un programa en C, existe una posibilidad de que la llamada retorne `NULL`. En tal caso esta semántica y la descrita por el estándar de C actúan de manera diferente.

Una opción para poder modelar una función de asignación de memoria que presente este tipo de comportamiento, es decidir de manera no determinística cuando esta función puede retornar `null`. El problema con esta opción es que complicaría el proceso de demostración de propiedades de un programa ya que cualquier llamada a la función de asignación de memoria dinámica podría fallar. Otra opción sería suponer que se cuenta con una cantidad fija de

memoria está disponible. Sin embargo, modelar este tipo de función no es una tarea trivial y permanece fuera del alcance de este trabajo.

Por lo tanto, en este trabajo se supone que se cuenta con una cantidad ilimitada de memoria y luego, cuando el proceso de traducción se lleva a cabo, se envuelve la función `malloc` de C en una función definida por el usuario que verifica si la llamada a `malloc` fue exitosa o no. Lo que se puede garantizar sobre un programa generado es que o bien se generará y tanto la ejecución del mismo en el ambiente de Isabelle como la ejecución en la máquina producirán estados finales que son equivalentes o el programa abortará si un error por falta de memoria es encontrado.

3.1.2. Semántica

La semántica de una expresión es su valor y el efecto que evaluar la misma tiene sobre el estado del programa. Para expresiones tales como $21 + 21$, la evaluación de la misma es trivial (42). Por otra parte, cuando se tienen expresiones con variables, tales como $foo + 42$, entonces se depende del valor de la variable. Por lo tanto se deben conocer el valor de una variable al momento de ejecución. Estos valores se almacenan en el estado del programa.

El estado de un programa es realmente un poco mas complicado que lo que se presenta a continuación. Aunque la sección 3.6 se dedica exclusivamente a discutir los estados, se procede a describir en esta sección las partes el estado que son necesarias para discutir la semántica de las expresiones en Chloe.

Valuaciones Se define el tipo para una valuación de la siguiente manera:

```
type_synonym valuation = vname ⇒ val option option
```

Una valuación es una función que mapea un nombre de una variable a un valor. El tipo de retorno es `val option option`, lo que modela los siguientes estados para el valor de una variable: no definida, no inicializada y posee un valor. Por lo tanto, dado un nombre de una variable, esta función puede producir uno de los siguientes resultados:

- `None`, que representa una variable que no está definida.
- `Some None`, que representa una variable que está definida pero no ha sido inicializada.
- `Some` v, que representa una variable que está definida e inicializada y contiene el valor v.

Estados visibles Cuando se ejecuta un comando, el mismo solo puede *ver* cierta parte del estado. La parte de un estado que un comando puede ver es aquella que contiene las variables que son locales a la función que se está ejecutando, las variables globales y la memoria. Esta parte del estado es precisamente lo que se llama *estado visible*. Se puede definir un estado visible como la parte de un estado que un comando puede ver y modificar. El tipo definido para ellos es el siguiente:

```
type_synonym visible_state = valuation × valuation × mem
```

Un estado visible es una tupla que contiene una función de valuación para las variables locales, una función de valuación para las variables globales y la memoria dinámica del programa.

Ahora se puede introducir la semántica para las expresiones en Chloe. Como fue expresado antes, la semántica de una expresión es su valor y el efecto que tiene la misma sobre el estado de un programa, por lo tanto se definen dos funciones de evaluación, una que calcula el valor de una expresión y una que calcula el valor de una expresión del LHS. Estas funciones se definen de la siguiente manera:

```
eval   :: exp ⇒ visible_state ⇒ (val × visible_state) option
eval_l :: lexp ⇒ visible_state ⇒ (addr × visible_state) option
```

donde `eval`, dada una expresión y un estado visible, retornará el valor de esta expresión y el estado visible resultante de evaluar dicha expresión. La función `eval_l`, dada una expresión del LHS y un estado visible, retornará el valor de esta expresión (el cual debe ser una dirección) y el estado resultante luego de evaluar dicha expresión. Es importante tener en cuenta que el tipo de retorno de las funciones de evaluación es el tipo `option`. Esto es debido a que estas funciones pueden fallar. Un fallo puede ocurrir en cualquier momento al evaluar una expresión y si un fallo es encontrado entonces el mismo se propaga hasta que toda la evaluación de la expresión devuelva un valor `None` que indica un error en la evaluación.

La evaluación de una expresión puede fallar por diversas razones que incluyen, pero no están limitadas a, variables no definidas, operandos ilegales en operaciones, acceso a partes inválidas de memoria y *overflow*. Por lo tanto, si hay un error temprano en la semántica de evaluación de expresiones, será detectado y propagado como un valor `None` que indica un estado erróneo.

Las funciones `eval` y `eval_l` dependen de otras funciones definidas con el fin de calcular correctamente los valores de las expresiones. La definición de todas las funciones auxiliares

```
detect_overflow  :: int ⇒ val option
read_var         :: vname ⇒ visible_state ⇒ val option
plus_val         :: val ⇒ val ⇒ val option
subst_val        :: val ⇒ val ⇒ val option
minus_val        :: val ⇒ val option
div_towards_zero :: int ⇒ int ⇒ int
div_val          :: val ⇒ val ⇒ val option
mod_towards_zero :: int ⇒ int ⇒ int
mod_val          :: val ⇒ val ⇒ val option
mult_val         :: val ⇒ val ⇒ val option
less_val         :: val ⇒ val ⇒ val option
not_val          :: val ⇒ val option
to_bool          :: val ⇒ bool option
eq_val           :: val ⇒ val ⇒ val option
new_block        :: val ⇒ mem ⇒ (val × mem) option
load             :: addr ⇒ mem ⇒ val option
```

FIGURA 3.4: Funciones auxiliares para eval y eval_l

para `eval` y `eval_l` se encuentra en la figura 3.4. A excepción de `div_towards_zero` y `mod_towards_zero`, cada una de estas operaciones pueden fallar, por lo que su tipo de retorno es τ `option`. Los valores de este tipo son o `None` cuando la operación falla o `Some`(v) donde v es de tipo τ.

Las funcionalidades de las funciones auxiliares son las siguientes:

- La función `detect_overflow` detecta el *overflow* en enteros. Toma como parámetro un valor del tipo entero predefinido por Isabelle y chequea si existe *overflow*con los límites mostrados en la figura 3.2. Esta función fallará cuando se detecte *overflow*. Al ser ejecutada exitosamente, la función retorna el valor correspondiente al entero dado como parámetro.

- La función `read_var` calcula el valor de una variable. Esta función falla cuando el nombre de la variable dado como parámetro corresponde a una variable no definida. Al ser ejecutada exitosamente, la función retorna el valor de la variable. Con el fin de calcular el valor de dicha variable, esta función chequea la valuación de las variables locales en el estado visible y retorna el valor de la variable si se encuentra definida allí. En el caso en el que la variable no se encuentre definida en el alcance local, la función procederá a chequear el alcance global y retornará el valor de la variable.

- La función `plus_val` calcula el valor de la suma entre dos valores. Esta función falla cuando se detecta *overflow* o cuando operandos distintos a dos enteros o una dirección y un entero (en ese orden específico) son dados como parámetros a la función. Al ser ejecutada exitosamente con dos valores de tipo entero como parámetros, la función

retorna un valor entero correspondiente a la suma de los operandos. Al ser ejecutada exitosamente con una dirección y un entero como parámetros, la función retorna una dirección correspondiente a la suma del desplazamiento entero al valor original de la dirección.

- La función `subst_val` calcula el valor de una sustracción entre dos valores. Esta función falla cuando se detecta *overflow* o cuando operandos distintos a dos enteros o una dirección y un entero (en ese orden específico) son dados como parámetros a la función. Al ser ejecutada exitosamente con dos valores de tipo entero como parámetros, la función retorna un valor correspondiente a la resta de los operandos. Al ser ejecutada exitosamente con una dirección y un entero como parámetros, la función retorna una dirección correspondiente a la resta del desplazamiento entero al valor original de la dirección.

- La función `minus_val` calcula el valor de la operación de menos unario sobre un valor. Esta función falla cuando se detecta *overflow* o cuando un operando distinto a un entero es dado. Al ser ejecutada exitosamente, la función retorna un valor entero correspondiente al resultado de negar el valor dado como parámetro.

- La función `div_towards_zero` realiza la división entera con truncamiento hacia cero.

- La función `div_val` calcula el valor de la división entre dos valores. La función falla cuando se detecta *overflow* o una división por cero o cuando operandos distintos a dos enteros son dados como parámetros a la función. Al ser ejecutada exitosamente, la función retorna un valor entero correspondiente al resultado de la división entera entre los dos operandos.

- La función `mod_towards_zero` realiza la operación de módulo con truncamiento hacia cero.

- La función `mod_val` calcula el valor de la operación módulo entre dos valores. La función falla cuando se detecta *overflow* o una división por cero o cuando operandos distintos a dos enteros son dados como parámetros a la función. Al ser ejecutada exitosamente, la función retorna un valor entero correspondiente al resultado de la operación de módulo entre los dos operandos.

- La función `mult_val` calcula el valor de la multiplicación entre dos valores. La función falla cuando se detecta *overflow* o cuando operandos distintos a dos enteros son dados

como parametros a la función. Al ser ejecutada exitosamente, la función retorna un valor entero correspondiente a la multiplicación de los operandos de la función.

- La función `less_val` calcula el valor resultante de realizar la operación menor que entre dos valores. La función falla cuando operandos distintos a dos enteros son dados como parámetros a la función. Al ser ejecutada exitosamente, la función retorna un valor entero `I` 1 cuando el primer operando es menor que el segundo y un valor entero `I` 0 de lo contrario.

- La función `not_val` calcula el valor resultante de realizar la negación lógica sobre un valor. La función falla cuando un operando distinto a un entero es dado como parámetro. Al ser ejecutada exitosamente, la función retorna un valor entero `I` 1 cuando el operando dado es un entero de valor `I` 0 y retorna un valor entero `I` 0 cuando el operando dado es un entero de valor diferente de `I` 0.

- La función `to_bool` retorna un valor de tipo lógico predefinido en Isabelle dado un operando. La función se utiliza para calcular evaluación de corto circuito para las operaciones `And` y `Or`. La función falla cuando un operando distinto a un entero es dado como parámetro. Al ser ejecutada exitosamente, la función retorna `False` cuando el parámetro dado tiene un valor igual a `I` 0 y retorna `True` de lo contrario.

- La función `eq_val` calcula el valor resultante de realizar la comparación de igualdad entre dos valores. La función falla cuando operandos diferentes a dos enteros o dos direcciones son dados como parámetros. Al ser ejecutada exitosamente con dos valores enteros como parámetros, la función retorna un valor entero `I` 1 si ambos operandos son iguales y un valor entero `I` 0 de lo contrario. Al ser ejecutada exitosamente con dos direcciones como parámetros, la función retorna un valor entero `I` 1 si ambas direcciones son iguales y un valor entero `I` 0 de lo contrario. Dos direcciones son consideradas como iguales cuando ambos componentes de la tupla son iguales.

- Las funciones `new_block` y `load` son aquellas explicadas anteriormente en la sección 3.1.1 que se encargan de asignar un nuevo bloque de memoria y cargar un valor desde memoria, respectivamente.

```
type_synonym fname = string

datatype
  com = SKIP
      | Assignl lexp exp
      | Assign  vname exp
      | Seq     com  com
      | If      exp com com
      | While   exp com
      | Free    lexp
      | Return exp
      | Returnv
      | Callfunl lexp fname "exp list"
      | Callfun vname fname "exp list"
      | Callfunv fname "exp list"
```

FIGURA 3.5: Comandos en Chloe

3.2. Comandos

En la siguiente sección se discutirá la sintaxis y semántica de los comandos en Chloe, así como las funciones y los programas escritos en el lenguaje. Ademas, se discutirán algunas suposiciones que la semántica toma que causan restricciones en la misma.

3.2.1. Sintaxis

Chloe contiene los siguientes constructores: asignación, secuenciación, condicionales, ciclos, SKIP[1], liberación de memoria, instrucción `return` y funciones. Las expresiones son aquellas descritas en la sección anterior (3.1).

Aquí se procede a describir la *sintaxis abstracta* de los comandos en el lenguaje Chloe.

En la figura 3.5 se encuentra la definición del tipo de datos creado en Isabelle para los comandos, donde `lexp` y `exp` son las expresiones descritas en la sección anterior (3.1), `vname` representa los nombres de variables y `fname` representa los nombres de funciones.

Para la asignación se definen dos comandos diferentes, uno de ellos permite la asignación a una variable, mientras que el otro permite la asignación a una ubicación en memoria. Se necesitan ambos comandos dado que el dominio definido para las direcciones y los valores enteros es disjunto, por lo tanto, una dirección no puede representar un valor entero y viceversa.

También se tienen dos comandos de retorno, uno de ellos permite retornar de una función que posee un valor de retorno, mientras que el otro es para retornar de una función que no lo posee.

[1]El comando SKIP es el equivalente a noop ya que no realiza operación alguna. Se utiliza con el fin de poder expresar otros constructores sintácticos como lo es un condicional sin una rama ELSE

Finalmente, se tienen tres comandos diferentes para llamadas a funciones. Uno de ellos (`Callfunv`) es para funciones que no poseen un valor de retorno. Los otros dos dependen de lo que se haga con el valor de retorno de la función, si el valor de retorno debe ser asignado a una variable se utiliza el comando `Callfun` y si el retorno debe ser asignado a una celda en memoria se utiliza el comando `Callfunl`.

En Isabelle se define una sintaxis concreta, la cual facilita la escritura y lectura de comandos en Chloe. En la tabla 3.1 se introduce la sintaxis concreta suponiendo que se tiene x que representa nombres de variables, a que representa expresiones, c, c_1 y c_2 que representan comandos, y que representa expresiones del LHS y f que representa nombres de funciones. A lo largo del trabajo se continuará utilizando la sintaxis concreta para facilitar la legibilidad.

Sintaxis abstracta	**Sintaxis concreta**
`Assignl` y a	y `::==` a
`Assign` x a	x `::=` a
`If` a c_1 c_2	`IF` a `THEN` c_1 `ELSE` c_2
`While a` c	`WHILE` a `DO` c
`Free` y	`FREE` y
`Return` a	`RETURN` a
`Returnv`	`RETURNV`
`Calllfunl` y f $[a]$	y `::==` f `(` $[a]$ `)`
`Calllfun` x f $[a]$	x `::=` f `(` $[a]$ `)`
`Calllfunv` f $[a]$	`CALL` f `(` $[a]$ `)`

Tabla 3.1: Equivalencia entre sintaxis abstracta y concreta

3.3. Funciones

En Chloe se tienen funciones que devuelven valores y aquellas que no tienen valor de retorno. Para las funciones que devuelven un valor es necesario saber que ocurre con ese valor de retorno. Al retornar de una llamada a función, el valor de retorno o bien debe ser asignado a una variable o a una celda en memoria o debe ser ignorado. En esta sección no se entrará en detalles para explicar esta decisión de diseño sino que se retrasará hasta la sección 3.7.1 donde se podrá explicar el razonamiento detrás de esta decisión de una manera mas adecuada.

Como es visible en la definición de la figura 3.6, una función consiste en un nombre, los parámetros formales, las variables locales y el cuerpo de la función, que es un comando, potencialmente grande, en el lenguaje Chloe. También se define un predicado que comprueba

```
record fun_decl =
  name :: fname
  params :: vname list
  locals :: vname list
  body :: com

valid_fun_decl :: fun_decl ⇒ bool
```

FIGURA 3.6: Definiciones de funciones

si una declaración de una función es válida o no. La declaración de una función se considera válida si y solo si los parámetros de la función y las variables locales tienen nombres distintos.

3.4. Programas

Un programa en Chloe consiste en un nombre, una lista de variables globales y una lista de funciones. En el apéndice A se encuentra la definición de un programa y su definición de validez.

Un programa se considera válido si cumple con todas las siguientes condiciones:

- Los nombres de las variables globales son distintos entre si.
- Los nombres de las funciones en el programa son diferentes entre si.
- Cada declaración de función para cada función en el programa debe ser válida.
- La función main debe estar definida.
- Ninguno de los nombres de las variables o nombres de función en el programa debe ser una palabra clave reservada del lenguaje C o una palabra clave reservada para *testing*.[2]
- Las variables globales y los nombres de las funciones en un programa no pueden ser iguales.

3.5. Restricciones

Esta semántica hace una suposición con respecto a la máquina en la que el código va a ser ejecutado. Esta restricción está parametrizada y corresponde a la arquitectura de la

[2]Dado que se quiere generar código C de la semántica de Chloe, se debe garantizar que ni los nombres de las variables ni de las funciones son alguna de las palabras claves reservadas en C o alguna de las palabras claves reservadas que se utilizan como nombres de variables en el proceso de *testing*.

máquina en la que el código será ejecutado. Como se dijo anteriormente, la precisión de los valores enteros puede ser modificada con el fin de hacer que esta semántica sea compatible con diferentes arquitecturas, por ejemplo arquitecturas de 32 bits.

Las suposiciones hechas por esta semántica pueden ser cambiadas al cambiar el parámetro `int_width`, el cual cambiará automáticamente los límites superior e inferior que se tienen para enteros (estos límites se describen en la figura 3.2).

Con el fin de garantizar las suposiciones hechas por la semántica, más adelante en el proceso de generación de código, se generan aserciones que aseguran que las condiciones que se suponen se cumplan.

Otra restricción en la semántica es que solo funciona para un subconjunto de C donde la semántica es determinística y cualquier comportamiento indefinido se considera como un error en la semántica. Se pasará a un estado erróneo si la semántica encuentra comportamiento que sea considerado como indefinido por el documento de referencia del lenguaje C (ISO/IEC, 2007), un ejemplo de este tipo de comportamiento es *overflow* de enteros.

3.6. Estados

En lenguajes más simples, que solo admiten un conjunto limitado de características tales como asignación, secuenciación de instrucciones, condicionales, ciclos y valores enteros, la representación del estado consiste simplemente en una función que mapea los nombres de variables a valores. En Chloe ese no es el caso, al incluir funciones, memoria dinámica y apuntadores al conjunto de características admitidas, la representación del estado del programa deja de ser una simple función que mapea nombres de variables a valores. En esta sección se detallan los componentes de la representación de un estado. Anteriormente, en la sección 3.1.2 se explico el concepto de un "estado visible", aquí se aclara la diferencia entre ese estado visible y un estado real, también se detallan los componentes del estado de un programa.

3.6.1. Valuación

Como se menciono anteriormente, una valuación es simplemente una función que mapea nombres de variables a valores. El tipo de retorno para esta función es `val option option` que se encarga de modelar tres estados diferentes que puede tener una variable: no definida, no inicializada o posee un valor.

3.6.2. Pila de ejecución

Chloe admite llamadas a funciones, para poder hacerlo se debe mantener una pila de ejecución en el estado. Esta pila de ejecución (de ahora en adelante se referirá a la misma simplemente como la pila) consiste en una lista de marcos de pila. Cada uno de estos marcos de pila contiene información importante acerca de la llamada a la función actual.

En la figura 3.7 el tipo definido para un marco de pila es una tupla que contiene un comando en Chloe, una valuación y una ubicación de retorno. El comando de Chloe corresponde al cuerpo de la función a ejecutar. La valuación corresponde a las variables locales a la función que fue llamada. Finalmente, la ubicación de retorno puede ser una de las siguientes: una dirección, una variable o una ubicación de retorno inválida. Cuando una función retorna una variable pueden suceder varias cosas:

- El valor se asigna a una variable. Esto es indicado por la ubicación de retorno correspondiente a una variable.
- El valor se asigna a una celda en memoria. Esto es indicado por la ubicación de retorno correspondiente a una dirección.
- El valor es ignorado. Esto es indicado por la ubicación de retorno inválida.

También se utiliza la ubicación de retorno inválida para las funciones que no poseen un valor de retorno.

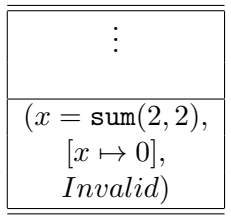
La ubicación de retorno se encuentra en el marco de pila del llamador, es decir, al retornar de una función el marco del llamador es el que debe ser revisado para saber a donde asignar el valor de retorno o si un valor de retorno es esperado. Para aclarar esto podemos tomar el ejemplo de código en la figura 3.8. Es un programa sencillo donde una función que suma el valor de sus dos parámetros se define y luego se llama desde la función main. Antes de la llamada a función, el marco de pila perteneciente a la función main tiene una ubicación de retorno *Invalid* y luego de la llamada a función la ubicación de retorno cambia a la variable x. Observe que el marco de pila que cambia su ubicación de retorno es aquel correspondiente a main, esto es porque el llamador es el que debe guardar la ubicación de retorno donde espera guardar el valor de retorno de una función llamada. Una ubicación de retorno *Invalid* indica que el llamador no espera guardar resultado alguno, es decir, el llamador no ha llamado aun a alguna función o la función llamada no tiene valor de retorno.

```
datatype return_loc = Ar addr | Vr vname | Invalid

type_synonym stack_frame = com × valuation × return_loc
```

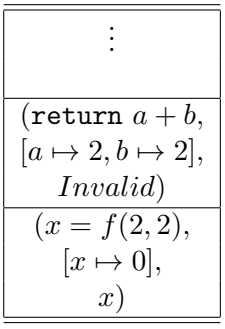
FIGURA 3.7: Definiciones relacionadas a la pila

```
int sum(int a; int b){
  return a+b;
}

int main(){
  int x = 0;
  x = sum(2,2);
}
```

(A) Código de ejemplo en C



(B) Pila antes de la llamada (crece hacia arriba)



(C) Pila luego de la llamada (crece hacia arriba)

FIGURA 3.8: Ejemplo de convención de llamada

3.6.3. Tabla de procedimientos

Otra extensión que debemos agregar al tratar con funciones es una tabla de procedimientos. Una tabla de procedimientos se define como sigue:

```
type_synonym proc_table = fname ⇀ fun_decl
```

donde "⇀ `fun_decl`" es equivalente a escribir "⇒ `fun_decl option`". Esta función mapea nombres de funciones a sus declaraciones.

Esta función se construye tomando la definición del programa y emparejando cada declaración de función en la lista a su nombre. Cada programa tiene su propia tabla de procedimientos.

3.6.4. Estado

Un estado se define como una tupla que contiene la pila, la valuación para las variables globales y la memoria dinámica.

```
type_synonym state = stack_frame list × valuation × mem
```

```
context fixes program :: program begin

  private definition proc_table ≡ proc_table_of program

  definition main_decl ≡ the (proc_table ''main'')
  definition main_local_names ≡ fun_decl.locals main_decl
  definition main_com ≡ fun_decl.body main_decl

  definition initial_stack :: stack_frame list where
    initial_stack ≡ [(main_com,
      map_of (map (λ. (x,None)) main_local_names),Invalid)]
  definition initial_glob :: valuation where
    initial_glob ≡ map_of (map (λ. (x,None)) (program.globals program))
  definition initial_mem :: mem where initial_mem ≡ []

  definition initial_state :: state where
    initial_state ≡ (initial_stack, initial_glob, initial_mem)

end
```

FIGURA 3.9: Construcción del estado inicial

3.6.5. Estado inicial

Con el fin de construir un estado inicial se deben definir ciertos componentes. En el apéndice A se encuentran definiciones para la configuración inicial de la pila, las variables globales y la memoria. La configuración inicial para la pila consiste en la pila que contiene únicamente el marco de pila para la función main del programa. La configuración inicial para la valuación global es una función donde cada posible nombre de variable se mapea al valor correspondiente a una variable no definida (`None`). La configuración inicial de la memoria dinámica es la memoria vacía, dado que nada ha sido asignado.

Luego de definir todos estos componentes, la configuración del estado inicial es dada por la tupla que contiene la configuración inicial de la pila, las variables globales y la memoria dinámica.

3.6.6. Estado visible

Adicionalmente, se debe definir un estado visible (como se mencionó anteriormente en la sección 3.1.2). Al ejecutar una función transformadora[3] sobre un estado (exceptuando las funciones transformadoras para las llamadas o retornos de funciones) la misma no podrá modificar alguna otra parte de la pila que no sea la valuación de variables locales del marco de pila actual. Se definen las transformaciones reales en el contexto de estados visibles y luego

[3]las funciones transformadoras se cubrirán más adelante en la sección 3.7

se levanta esta definición a estados. Por lo tanto, una función transformadora sobre estados, con excepción de las llamadas o retornos de funciones, no puede manipular la pila.

El marco de pila en el tope de la pila corresponde a la función actual que está siendo ejecutada y el comando en ese marco de pila corresponde al comando (o programa) en Chloe que está siendo ejecutado. Cada vez que se ejecuta un comando o se toma un paso en la semántica de pasos cortos, este comando se actualiza para contener el próximo comando a ejecutar. Con el fin de demostrar que la semántica de pasos cortos es determinística se debe demostrar que cada vez que se tiene una pila de ejecución no vacía, el orden en que se aplica el transformador de evaluación[4] y la función que actualiza el comando a ejecutar a continuación es irrelevante para el estado final resultante. Es por ello que se introduce una definición separada para un estado visible aparte de la de un estado regular, es una vista del mismo estado completo pero con información limitada.

3.7. Semántica de pasos cortos

3.7.1. CFG

Un Grafo de Control de Flujo (CFG por sus siglas en ingles *Control Flow Graph*) es una representación en forma de grafo que cubre los diferentes caminos que un programa puede tomar durante su ejecución. Se tiene el concepto de ubicación actual, que es un *program pointer* a un nodo. Un comando se ejecuta siguiendo una arista desde el nodo al que apunta el *program pointer* a un nuevo nodo. Los nodos del CFG son comandos. Las aristas del CFG contienen una anotación con dos funciones que dependen del estado actual del programa. La primera de estas funciones indica si una arista puede ser seguida o no (por ejemplo, en le caso de un condicional) y la segunda indica como será transformado el estado al seguir la arista, es decir, el efecto que tiene seguir la arista sobre el estado del programa. Estas funciones son llamadas *enabled* y *transformer*. A continuación se describen en detalle las definiciones para estas funciones.

Funciones *enabled* Una función *enabled* es una función parcial que se define como sigue:

```
type_synonym enabled = state ⇀ bool
```

[4]Este transformador es la función eval levantada para trabajar en estados en lugar de estados visibles, se discutirá con mayor detalle en la sección 3.7

```
fun truth_value_of :: val ⇒ bool where
  truth_value_of NullVal ⟷ False
| truth_value_of (I i) ⟷ i ≠ 0
| truth_value_of (A _) ⟷ True

abbreviation en_always :: enabled where en_always ≡ λ_. Some True

definition en_pos :: exp ⇒ enabled

definition en_neg :: exp ⇒ enabled
```

FIGURA 3.10: Funciones *enabled*

Indica si un estado está habilitado para continuar su ejecución. Esta es una función parcial, por lo que su ejecución podría fallar. La ejecución de una función *enabled* falla cuando encuentra un error en su evaluación y retorna un valor `None` que representa un estado erróneo.

Esta función es útil para la ejecución de construcciones condicionales en el lenguaje.

Si se tiene el término "`IF` b `THEN` c_1 `ELSE` c_2". Cuando la evaluación de la condición b produce un valor `True` se seguirá aquella arista que lleva al nodo que contiene c_1. Mientras que cuando la evaluación de b produce un valor `False` se seguirá aquella arista que lleva al nodo que contiene c_2. Dependiendo del resultado de la función habilitada se decide que arista puede seguir el nodo. El único caso donde una ejecución no podría continuar es cuando no hay ninguna arista habilitada que seguir. Afortunadamente, esto no puede pasar en los programas de Chloe ya que siempre hay una arista habilitada. A excepción del condicional, para cada comando en Chloe la función *enabled* siempre retorna `True`. En el caso de un condicional, la ejecución puede continuar o bien siguiendo la arista hasta el primer comando o siguiendo la arista hasta el segundo comando. Siempre habrá una arista que puede ser seguida luego de un nodo que contenga un condicional.

Una lista de las funciones *enabled* que son utilizadas se presenta en la figura 3.10. La función `truth_value_of` mapea un valor a un valor de tipo lógico, es decir `True` o `False`. También se encuentran las funciones `en_always` que siempre retorna `True`, `en_pos` que solo retorna `True` cuando el valor real (calculado con `truth_value_of`) de una expresión dada como parámetro se evalúa como `True` y `en_neg` que solo retorna `True` cuando el valor real (calculado con `truth_value_of`) de una expresión dada como parámetro se evalúa como `False`. Estas funciones serán utilizadas mas tarde en la definición del CFG.

Funciones *transfomer* Una función *transformer* es una función parcial que se define como sigue:

```
type_synonym transformer = state ⇀ state
```

Es una función parcial que realiza una transformación de un estado a otro. Dado que es una función parcial, su ejecución puede fallar. La ejecución de una función *transformer* falla cuando se encuentra un error en algún momento de su ejecución y retorna un valor `None` que indica un estado erróneo.

Se definen funciones que producen una función *transformer* para cada comando en Chloe. Estas funciones serán utilizadas mas tarde en la definición del CFG y se definen en la figura 3.11. Se define una función *transformer* `tr_id` que sirve como la función identidad y simplemente retorna el mismo estado que recibe como parámetro.

Las definiciones presentadas en la figura 3.11 producen una función *transformer* que será utilizada al ejecutar un comando. Se procede a describir de una manera aproximada el efecto que tendrán estas funciones *transformer* producidas al ser aplicadas a estados.

En primer lugar se tiene la función *transformer* para una asignación producida por `tr_assign`, se encarga de evaluar la expresión que se quiere asignar y luego realizar una operación `write` al estado. Retorna el estado resultante de realizar dicha evaluación y la operación `write`.

La función *transformer* para una asignación a una celda en memoria producida por `tr_assignl`, se encarga de evaluar la expresión del LHS para obtener la ubicación de memoria donde se guardará el valor, evaluar la expresión para obtener el nuevo valor y guardar el valor en memoria. Retorna el estado resultante de realizar dichas evaluaciones y la operación `store`.

La función *transformer* para una evaluación producido por `tr_eval`, se encarga de evaluar una expresión. Retorna el estado resultante de realizar dicha evaluación.

La función *transformer* para una operación de liberación de memoria producido por `tr_free`, se encarga de evaluar la expresión del LHS que recibe como parámetro con el fin de obtener una dirección y liberar el bloque correspondiente a esta dirección en memoria. Retorna el estado resultante de realizar dicha evaluación y la liberación de memoria.

Cuando se realiza una llamada a función se debe chequear que los parámetros formales y los parámetros reales dados a la función tengan el mismo tipo y que cada parámetro formal tenga un parámetro real correspondiente. Dado que no se posee un sistema de tipos estático solamente se chequeara la segunda condición mencionada. Ademas, el orden de evaluación de los parámetros dados a una función es de izquierda a derecha. Los parámetros siempre se evalúan siguiendo un orden de derecha a izquierda. Finalmente, cuando se llama a una

función también se deben mapear los parámetros formales a los valores de los parámetros dados y considerarlos como variables locales en el alcance de la función.

Se tiene una función `call_function` que produce una función *transformer* para cualquier llamada a función, esta verifica que el número de parámetros formales y parámetros dados sea el mismo, evalúa los parámetros dados de izquierda a derecha, crea un nuevo marco de pila que contiene el cuerpo de la función, la valuación de las variables locales (la cual incluye los parámetros mapeados a sus valores dados y las variables locales mapeadas al valor que representa no inicializado) y la ubicación de retorno `Invalid`, retorna el estado resultante de realizar dichas operaciones sobre el estado.

Cuando se llama a una función, el llamador tiene que cambiar su marco de pila con el fin de actualizar el valor de la ubicación de retorno en el marco de pila actual. Se definen diferentes funciones que realizan ese cambio dependiendo del tipo de llamada a función y producen una función *transformer* llamando a `call_function`.

Las diferentes funciones que se definen son `tr_callfunl`, `tr_callfun` y `tr_callfunv`. Se utiliza `tr_callfunl` cuando la ubicación de retorno de una función es una celda en memoria. En este caso, la función evalúa primero la expresión del LHS para obtener la dirección de retorno y la utiliza para actualizar la ubicación de retorno del marco de pila antes de llamar a `call_function`. La función *transformer* resultante retornará el estado resultante de realizar esta evaluación, actualizar el marco de pila y las operaciones hechas por `call_function`. Es importante resaltar que la evaluación de la expresión del LHS que retorna una dirección debe ser la primera que se realiza, esto es con el fin de evitar comportamiento no deseado, ya que la función llamada podría cambiar el estado.

Se utiliza `tr_callfun` cuando la ubicación de retorno de la función es una variable. En este caso la función actualiza la ubicación de retorno del marco de pila con el nombre de la variable antes de llamar a `call_function`, la función *transformer* resultante retornará el estado resultante de actualizar el marco de pila y las operaciones hechas por `call_function`.

Se utiliza `tr_callfunl` cuando no se espera que la función llamada retorne un valor. En este caso, la función actualiza la ubicación de retorno del marco de pila con `Invalid` antes de llamar a `call_function`, la función *transformer* resultante retorna el estado resultante de actualizar el marco de pila y las operaciones hechas por `call_function`.

Se tiene una función `tr_return` que produce una función *transformer* para una llamada a `return` que retorna una expresión de una función, esta desempila el marco de pila en el tope de la pila (el cual pertenece a la función de la que se retorna), evalúa el valor correspondiente

```
abbreviation (input) tr_id :: transformer where tr_id ≡ Some

tr_assign :: vname ⇒ exp ⇒ transformer
tr_assignl :: lexp ⇒ exp ⇒ transformer
tr_eval :: exp ⇒ transformer
tr_free :: lexp ⇒ transformer
call_function :: proc_table ⇒ fname ⇒ exp list ⇒ transformer
tr_callfunl :: proc_table ⇒ lexp ⇒ fname ⇒ exp list ⇒ transformer
tr_callfun :: proc_table ⇒ vname ⇒ fname ⇒ exp list ⇒ transformer
tr_callfunv :: proc_table ⇒ fname ⇒ exp list ⇒ transformer
tr_return :: exp ⇒ transformer
tr_return_void :: transformer
```

FIGURA 3.11: Funciones *transformer*

a la expresión retornada y, si la pila no está vacía, busca el valor de retorno y dependiendo de si es una dirección, una variable o una ubicación `Invalid` retorna el estado resultante de guardar el valor en memoria, guardar el valor en una variable o retornar el estado como estaba, respectivamente. Nótese que si la función retorna un valor pero su ubicación de retorno es `Invalid` entonces el valor retornado es ignorado en lugar de ser considerado como una ejecución errónea.

Finalmente, se tiene una función `tr_return_void` que produce una función *transformer* para una llamada a `return` en una función que no retorna un valor. Esta función *transformer* se encarga de desempilar el marco de pila en el tope de la pila (el cual pertenece a la función de la que se retorna). Posteriormente, si la pila no está vacía, obtiene la ubicación de retorno. Si la ubicación de retorno es una ubicación diferente de `Invalid`, retorna un valor `None` que indica un estado erróneo. Sin embargo, si la ubicación de retorno es, en efecto, `Invalid` entonces retorna el estado resultante de desempilar el último marco de pila.

CFG Para poder hablar de ejecuciones siguiendo las aristas del CFG se debe introducir un nuevo concepto. Si se toma la definición de un CFG dada al inicio de esta sección, se tiene que para poder hablar de una ejecución de un comando siguiendo las aristas del CFG se debe tener un *program pointer* que apunte al nodo actual. También se debe introducir el concepto de una pila. Se tiene una pila de *program pointers* que acompaña el CFG y se desempilará un nuevo *program pointer* de la pila una vez que se haya *seguido* el *program pointer* actual hasta un nodo que contenga `SKIP`. Un *program pointer* al cuerpo de una función se empilará cuando una llamada a función sea realizada. Un *program pointer* se desempilará de la pila cuando exista una instrucción `return`. Este concepto será importante al hablar de llamadas a funciones y retornos de funciones más adelante.

En el apéndice B se encuentra una definición inductiva para CFG, donde se pueden ver las reglas para formar aristas entre comandos. Se puede seguir una arista en el CFG desde una asignación a una variable a `SKIP`. Esta arista siempre está habilitada (la función *enabled* retorna `True`) y la función *transformer* para actualizar el estado es el resultado de llamar a `tr_assign` con los parámetros específicos para el comando.

Del mismo modo, una arista del CFG puede ser seguida desde una asignación a una celda en memoria hasta `SKIP`, esta arista siempre está habilitada (la función *enabled* retorna `True`) y la función *transformer* para actualizar el estado es el resultado de llamar a `tr_assignl` con los parámetros específicos para el comando.

Para la secuenciación de instrucciones existen dos aristas diferentes que pueden ser seguidas. Cuando el primer comando en la secuenciación es `SKIP`, se puede seguir una arista que va desde el nodo con el comando completo hasta el nodo que tiene solamente el segundo comando. Esta arista siempre está está habilitada (la función *enabled* retorna `True`) y la función *transformer* para actualizar el estado es `tr_id`. El segundo caso ocurre cuando se tiene un comando de la forma `(` c_1 `;;` c_2 `)` donde c_1 no es `SKIP`. Si es posible seguir una arista desde c_1 hasta otro nodo c_1' (donde dicha arista está anotada con a, que es una tupla que contiene una función *enabled* y una función *transformer*) entonces es posible seguir una arista (también anotada con a) desde `(` c_1 `;;` c_2 `)` hasta `(` c_1' `;;` c_2 `)`.

Para un condicional también se tienen dos casos. Se tendrá la posibilidad de seguir una arista hacia el primer comando u otra arista hacia el segundo comando, dependiendo del valor de la condición. En el caso donde se sigue la arista que lleva hasta el primer comando, se va desde `IF` b `THEN` c_1 `ELSE` c_2 hasta c_1. Esta arista estará anotada con una función *enabled* dada por `en_pos` que solo habilitará esta arista cuando la condición b se evalúe a `True` y una función *transformer* dado por `tr_eval`. En el caso donde se sigue la arista que lleva hasta el segundo comando, se va desde `IF` b `THEN` c_1 `ELSE` c_2 hasta c_2. Esta arista estará anotada con una función *enabled* dada por `en_neg` que solo habilitará esta arista cuando la condición b se evalúe a `False` y una función *transformer* dado por `tr_eval`.

En el caso de un ciclo siempre se puede seguir una arista que va desde `WHILE` b `DO` c hasta `IF` b `THEN` c `;; WHILE` b `DO` c `ELSE SKIP` al expandir la definición del ciclo una vez. La arista tendrá una función *enabled* que siempre está habilitada y la función *transformer* para el estado es `tr_id`.

Se puede seguir una arista desde `FREE` X hasta `SKIP`. Esta arista siempre está *enabled* y tiene una función *transformer* dado por `tr_free` llamado con x.

Se puede seguir una arista desde cualquiera de los dos comandos de retorno existentes hasta `SKIP`. Esta arista siempre está habilitada (la función *enabled* retorna `True`) y tiene una función *transformer* dado como resultado de `tr_return` o `tr_return_void` dependiendo de que comando de retorno sea. Nótese que al *ejecutar* este comando en el CFG, seguir una arista desde un nodo que contiene un retorno hasta `SKIP` desempilará el *program pointer* que apunta al nodo `SKIP` y continuará la "ejecución" en el nodo apuntado por el próximo *program pointer* en la pila.

Finalmente, se puede seguir una arista desde cualquiera de los nodos de llamada a función hasta `SKIP`. Esta arista siempre está habilitada (la función *enabled* retorna `True`) y tiene una función *transformer* dado como resultado de `tr_callfunl`, `tr_callfun` o `tr_callfunv` dependiendo de si es una llamada a función que retorne a una celda en memoria, una variable o sin retorno. Nótese que, también en este caso, al *ejecutar* este comando en el CFG, seguir la arista desde una llamada a función hasta un nodo `SKIP` empilará un nuevo *program pointer* que apunte al nodo con el cuerpo de la función y continuará la "ejecución" en el nodo apuntado por el próximo *program pointer* en la pila.

3.7.2. Reglas de la semántica de pasos cortos

Finalmente, se introducen las reglas para la semántica de pasos cortos de Chloe. Se prefiere una semántica de pasos cortos en lugar de una de pasos largos dado que se quiere una semántica mas detallada. La semántica de pasos largos tiene un gran inconveniente y es que no puede diferenciar entre una ejecución que no termina y una que se queda atascada en una configuración errónea. Es por ello que se prefiere una semántica mas detallada que permita diferenciar entre no terminación y quedarse atascado en un estado erróneo, dado que permite hablar de estados intermedios durante la evaluación.

Por lo general una configuración en una semántica de pasos cortos es un par que contiene un comando y un estado. Dado que se trabaja con funciones y se tiene el comando que se esta ejecutando en el marco de pila, la definición de pasos cortos dada en este trabajo es dada sobre estados. Un paso pequeño y atómico puede ser tomado desde un estado a otro.

Las reglas para la semántica de pasos cortos se encuentran detalladas en la figura 3.12. La notación de infijo para la semántica de pasos cortos se escribe: $s \to s_2$ lo cual significa que se toma un paso corto desde s hasta s_2. Se puede dar un paso cuando las siguientes condiciones se cumplen:

- La pila no está vacía

- Hay una arista en el CFG entre c_1 y c_2.
- El comando en el marco de pila en el tope de la pila en el estado inicial es c_1.
- Al aplicar la función *enabled* sobre el estado, esta retorna `True`.
- Al aplicar la función *transformer* al estado con el comando en marco de pila en el tope de la pila actualizado de c_1 a c_2, este retorna un nuevo estado s_2.

Si todas las condiciones mencionadas anteriormente se cumplen, entonces se puede tomar un paso corto desde el estado s hasta s_2.

Un paso corto también puede ser tomado al retornar de una función que no retorne valor alguno. Si el comando en el tope de la pila es `SKIP`, la pila no está vacía y aplicar la función *transformer* en el estaod inicial s produce un nuevo estado s_2, entonces podemos tomar un paso corto desde s hasta s_2.

El tipo para `small_step` es de `state` a `state option`. La segunda configuración está encerrada en un tipo opción ya que tomar un paso corto puede resultar en una estado erróneo donde la ejecución se quedará atascada.

Tomar un paso corto puede fallar en cualquiera de los siguientes casos:

- La función *enabled* o la función *transformer* retornan `None` al ser evaluados sobre el estado inicial. Esto indica que un estado erróneo fue alcanzado durante la evaluación de alguna de esas dos funciones. El estado erróneo se propaga al tomar un paso corto desde el estado s hasta `None` y luego la ejecución se queda atascada allí.
- Si al aplicar la función *transformer* `tr_return_void` sobre el estado el mismo retorna un valor `None`, el comando en el tope de la pila en el estado inicial es SKIP y la pila no está vacía, entonces también se propaga este estado erróneo `None` al tomar un paso corto desde s hasta `None`. Esto indica que hubo un error al retornar de una función sin valor de retorno.

Se ha definido como tomar un solo paso en la semántica. Con el fin de tomar mas de un solo paso y definir la ejecución de un programa en la semántica se debe levantar la definición de `small_step` a `state option` tanto en el estado inicial como en el final.

También se define (en la figura 3.12) una nueva definición `small_step'` basada en la definición previa para `small_step`. Esta definición básicamente dice que si un paso corto puede ser tomado desde el estado s hasta s' entonces un paso corto puede ser tomado desde

```
inductive
  small_step :: state ⇒ state option ⇒ bool (infix → 55)
where
  Base: ⟦ ¬ is_empty_stack s; c1=com_of s; cfg c1(en,tr)c_{2};
    en s = Some True; tr (upd_com c2 s) = Some s2 ⟧ ⟹ s → Some s2
| None: ⟦ ¬ is_empty_stack s; c1=com_of s; cfg c1(en,tr)c_{2};
    en s = None ∨ tr (upd_com c2 s) = None⟧ ⟹ s → None
| Return_void: ⟦ ¬ is_empty_stack s; com_of s = SKIP;
    tr_return_void s = Some s' ⟧ ⟹ s → Some s'
| Return_void_None: ⟦ ¬ is_empty_stack s; com_of s = SKIP;
    tr_return_void s = None ⟧ ⟹ s → None

inductive
  small_step' :: (state) option ⇒ (state) option ⇒ bool (infix →'  55)
where
  s → s' ⟹ Some s →' s'

abbreviation
  small_steps :: (state) option ⇒ (state) option ⇒ bool (infix →* 55)
where s0 →* sf ==  star small_step' s0 sf
```

FIGURA 3.12: Reglas de la semántica de pasos cortos

`Some` s hasta s'. Nótese que s' puede ser tanto `None` como `Some` s_i. La notación de infijo para esta nueva definición se escribe como `Some` $s \to' s'$, lo cual significa que se toma un paso corto desde `Some` s hasta s'.

De esta manera, se ha levantado la definición de pasos cortos al tipo `state option` y se puede definir la ejecución de un programa como la clausura reflexiva y transitiva de `small_step'`, usando el operador `star` de Isabelle. La notación de infijo para esto es escrito: $s_0 \to *s_f$, lo cual significa que se puede ir desde el estado s_0 hasta s_f en cero o mas pasos cortos.

Determinísmo

Para poder ejecutar la semántica dentro del ambiente de Isabelle/HOL, la misma debe ser determinística. Es por ello que se demuestra que la semántica es determinística. Con el fin de demostrar esa propiedad, una serie de lemas deben ser definidos y demostrados primero.

Cada vez que se introduce un lema, junto a su número se encontrará un nombre en paréntesis. Este nombre corresponde al nombre del lema en el código fuente de Isabelle presentados con este trabajo. En el caso en que el lector esté interesado en la demostración exacta para un lema en particular él o ella puede buscar el lema correspondiente en los archivos y leer la prueba.

Dado que las demostraciones de los siguientes lemas no son el enfoque principal de este trabajo, las mismas se presentan en el apéndice C

Este primer lema indica que cada vez que se tiene un comando diferente a `SKIP` hay siempre una arista habilitada (la función *enabled* retorna `True`) que se puede tomar en el CFG o un error ocurre al evaluar la función *enabled*.

Lema 3.1 (cfg_has_enabled_action)**.**
$c \neq \texttt{SKIP} \Longrightarrow \exists\ c'\ en\ tr.\ \texttt{cfg}\ c\ (en, tr)\ c'\ \wedge\ (en\ s\ =\ \texttt{None} \vee en\ s\ =\ \texttt{Some True})$

A continuación se quiere demostrar que mientras la pila no esté vacía, la semántica de pasos cortos siempre puede tomar un paso. Se utilizará el lema anterior en la prueba para este nuevo lema. Este lema dice que la semántica puede tomar un paso.

Lema 3.2 (can_take_step)**.**
$\neg\ \texttt{is\_empty\_stack}\ s \Longrightarrow \exists\ x.\ s\ \rightarrow\ x$

Se definen varios lemas que serán útiles más adelante.

El primer lema establece que el CFG se queda atascado en SKIP:

Lema 3.3 (cfg_SKIP_stuck)**.**
$\neg\ \texttt{cfg SKIP}\ a\ c$

Lema 3.4 (ss_empty_stack_stuck)**.**
$\texttt{is\_empty\_stack}\ s \Longrightarrow \neg\ (s\ \rightarrow\ cs')$

Lema 3.5 (ss'_SKIP_stuck)**.**
$\texttt{is\_empty\_stack}\ s \Longrightarrow \neg\ (Somes\ \rightarrow\ cs')$

Los lemas 3.4 y 3.5 definen el estado final en el que la semántica se quedará atascada, la semántica se quedará atascada cuando no existan mas marcos de pila en la pila.

Lema 3.6 (en_neg_by_pos)**.**
$\texttt{en\_neg}\ e\ s\ =\ \texttt{map\_option (HOL.Not) (en\_pos}\ e\ s\ \texttt{)}$

El lema 3.6 establece que cada vez que la función `en_neg` tenga un valor (diferente de None) sobre un estado, `en_pos` tendrá el mismo resultado opuesto. En el caso donde una de las funciones falla, la otra fallará también y retornarán `None`. Si no fallan, sus resultados serán opuestos, esto es, si una tiene `Some True` como resultado, la otra tendrá `Some False` como resultado. Este lema será útil al probar que la semántica es determinística.

Lema 3.7 (cfg_determ)**.**
$\texttt{cfg}\ c\ a1\ c' \wedge \texttt{cfg}\ c\ a2\ c''$

$\Longrightarrow a1 \; = \; a2 \wedge c' \; = \; c'' \vee$
$\exists\, b.\; a1 \; = (\texttt{en\_pos}\; b,\; \texttt{tr\_eval}\; b) \; \wedge \; a2 \; = (\texttt{en\_neg}\; b,\; \texttt{tr\_eval}\; b) \vee$
$\exists\, b.\; a1 \; = (\texttt{en\_neg}\; b,\; \texttt{tr\_eval}\; b) \; \wedge \; a2 \; = (\texttt{en\_pos}\; b,\; \texttt{tr\_eval}\; b)$

El lema 3.7 establece que CFG es determinístico. El único caso donde no lo es es en el caso del condicional, para el cual se agrega una alternativa extra en la conclusión. Puede pasar que se tenga una regla del CFG comenzando en un comando `If` que tiene una arista a c_1 y también una arista a c_2. Esto no es realmente un problema dado que las funciones *enabled* garantizan que cuando esto ocurre, solamente una de las aristas puede ser tomada.

Lema 3.8 (lift_upd_com)**.**
$\neg\, \texttt{is\_empty\_stack}\; s \; \Longrightarrow$
$\texttt{lift\_transformer\_nr}\; tr\; (\texttt{upd\_com}\; c\; s) \; =$
$\texttt{map\_option}\; (\texttt{upd\_com}\; c)\; (\texttt{lift\_transformer\_nr}\; tr\; s)$

Lema 3.9 (tr_eval_upd_com)**.**
$\neg\, \texttt{is\_empty\_stack}\; s \; \Longrightarrow$
$\texttt{tr\_eval}\; e\; (\texttt{upd\_com}\; c\; s) \; =$
$\texttt{map\_option}\; (\texttt{upd\_com}\; c)\; (\texttt{tr\_eval}\; e\; s)$

La función `lift_transformer_nr` levanta la definición de una función *transformer*, que opera en el nivel de estados visibles, a estados. El lema 3.8 establece que no importa en que orden se aplique una función *transformer* sobre un estado y se actualice el comando en el tope de la pila, dado que siempre produce el mismo resultado. Esto es porque la función que actualiza el comando solo modifica el comando en el tope de la pila y la función *transformer* no puede acceder y modificar esa parte de la pila, ya que solo puede modificar el estado visible.

El lema 3.9 es una versión más específica del lema anterior que establece el mismo hecho específicamente para la función `tr_eval`.

Todos los lemas y definiciones anteriores son definidos con el fin de ser utilizados en la próxima demostración; la semántica de pasos cortos es determinística.

Lema 3.10 (small_step_determ)**.**
$s \; \rightarrow \; s' \; \wedge \; s \; \rightarrow \; s'' \; \Longrightarrow \; s' \; = \; s''$

Luego solo queda demostrar que `small_step'` es también determinística.

Lema 3.11 (small_step'_determ)**.**
$s \; \rightarrow' \; s' \; \wedge \; s \; \rightarrow' \; s'' \; \Longrightarrow \; s' \; = \; s''$

```
datatype cfg_edge = Base transformer com
                  | Cond enabled transformer com com

context fixes proc_table :: proc_table begin

  fun cfg_step :: "com ⇒ cfg_edge" where
    "cfg_step SKIP = undefined"
  | "cfg_step (x ::= a) = Base (tr_assign x a) SKIP"
  | "cfg_step (x ::== a) = Base (tr_assignl x a) SKIP"
  | "cfg_step (SKIP;; c2) = Base tr_id c2"
  | "cfg_step (c1;;c2) = (case cfg_step c1 of
      Base tr c ⇒ Base tr (c;;c2)
    | Cond en tr ca cb ⇒ Cond en tr (ca;;c2) (cb;;c2)
     )"
  | "cfg_step (IF b THEN c1 ELSE c2) = Cond (en_pos b) (tr_eval b) c1 c2"
  | "cfg_step (WHILE b DO c) =
      Base tr_id (IF b THEN c;; WHILE b DO c ELSE SKIP)"
  | "cfg_step (FREE x) = Base (tr_free x) SKIP"
  | "cfg_step (Return a) = Base (tr_return a) SKIP"
  | "cfg_step Returnv = Base (tr_return_void) SKIP"
  | "cfg_step (Callfunl e f params) =
      Base (tr_callfunl proc_table e f params) SKIP"
  | "cfg_step (Callfun x f params) =
      Base (tr_callfun proc_table x f params) SKIP"
  | "cfg_step (Callfunv f params) =
      Base (tr_callfunv proc_table f params) SKIP"

end
```

FIGURA 3.13: Aristas de un solo paso

```
definition fstep :: proc_table ⇒ state ⇒ state option where
  fstep proc_table s ≡
    if com_of s = SKIP then
      tr_return_void s
    else
      case cfg_step proc_table (com_of s) of
          Base tr c' ⇒ tr (upd_com c' s)
        | Cond en tr c1 c2 ⇒ do {
            b ← en s;
            if b then
              tr (upd_com c1 s)
            else
              tr (upd_com c2 s)
          }
```

FIGURA 3.14: Definición de fstep

3.8. Interpretador

3.8.1. Ejecución de un solo paso

Existen dos tipos de pasos que se pueden tomar en el CFG. Se crea un nuevo tipo de datos par representarlos.

Un paso *Base* tiene una función *transformer* y siempre está habilitada para seguir hasta

un nuevo comando y un paso *Cond* que, además de la función *transformer*, también tiene una función *enabled* y dos comandos. Esta función *enabled* indica si se debe tomar un paso al primer o al segundo comando. También se define una función `cfg_step` que dada el comando de inicio retorna que tipo de paso sigue en el CFG.

La función `fstep`, definida en la figura 3.14, indica como se toma un solo paso el la ejecución de la semántica. La ejecución de un paso en la semántica va desde un estado inicial a un nuevo estado que puede ser erróneo (`None`) o uno válido (`Some` s). Para ejecutar un comando `SKIP` se llama a `tr_return_void`. En cualquier otro comando se verifica que tipo de paso se debe tomar utilizando la función `cfg_step` y, basado en esto, se decide que hacer. Si es un paso `Base` se llama a la función *transformer* sobre el estado con el comando actualizado. Si es un paso `Cond` se evalúa la condición y se llama a la función *transformer* sobre el estado con el comando actualizado.

Equivalencia entre semántica de pasos cortos y ejecución de un solo paso

Ahora se debe demostrar que le ejecución de un solo paso es semánticamente equivalente a tomar un paso en la semántica de pasos cortos. Esto significa probar que $\neg$ `is_empty_stack` $\Longrightarrow$ $s \rightarrow s' \longleftrightarrow$ `fstep` $s = s'$ [5]. Se demuestran ambas direcciones de la equivalencia por separado: para cada paso tomado en la semántica de pasos cortos hay un paso equivalente que puede ser tomado en la ejecución por `fstep` que llevará al mismo estado final y viceversa.

Se comienza demostrando que cualquier paso tomado en la semántica de pasos cortos puede ser simulado a través de un paso tomado con `fstep`.

Lema 3.12 (fstep1)**.**
$s \rightarrow s' \Longrightarrow$ `fstep` $s = s'$

Luego se considera la dirección opuesta:

Lema 3.13 (fstep2)**.**
$\neg$ `is_empty_stack` $s \Longrightarrow s \rightarrow$ (`fstep` s)

Ambas direcciones juntas (lema 3.12 y lema 3.13) permiten demostrar la equivalencia planteada al inicio:

Lema 3.14 (ss_fstep_equiv)**.**
$\neg$ `is_empty_stack` $\Longrightarrow s \rightarrow s' \longleftrightarrow$ `fstep` $s = s'$

[5] fstep tiene un parámetro extra, es decir la tabla de procedimientos, que no se escribirá acá para simplificar la lectura.

```
fun is_term :: "state option ⇒ bool" where
  "is_term (Some s) = is_empty_stack s"
| "is_term None = True"

definition interp :: "proc_table ⇒ state ⇒ state option" where
  "interp proc_table cs ≡ (while
    (HOL.Not ∘ is_term)
    (λSome cs ⇒ fstep proc_table cs)
    (Some cs))"
```

FIGURA 3.15: Definición de un interpretador para Chloe

3.8.2. Ejecución e interpretación

Al igual que en la sección anterior las demostraciones para los lemas de esta sección se encuentran en el apéndice D.

Con el fin de ejecutar un programa se define un interpretador.

La definición para tal interpretador se encuentra en la figura 3.15. Primero se define el criterio para considerar un estado como final. Un estado será considerado como final cuando su pila de ejecución esté vacía o cuando sea `None`.

El interpretador para la semántica funciona de la siguiente manera: mientras no se alcance un estado final se ejecuta `fstep`.

Se presenta un lema que establece que si un estado es final, entonces es el resultado de la interpretación.

Lema 3.15 (interp_term)**.**
`is_term` (`Some` cs) $\Longrightarrow$ `interp proc_table` cs $=$ `Some` cs

Para poder demostrar esto necesitamos un lema que expanda la definición del ciclo en la definición de nuestro interpretador:

Lema 3.16 (interp_unfold)**.**
`interp proc_table` cs = (
`if is_term` (`Some` cs) `then Some` cs
`else do`{ $cs \leftarrow$ `fstep proc_table` cs; `interp proc_table` cs })

Con el lema 3.16, la demostración del lema 3.15 se hace automáticamente.

Solamente programas válidos pueden ser ejecutados. En la figura 3.16 se encuentra la definición de la función que ejecuta un programa. Para ejecutar un programa se debe asegurar que el mismo es válido utilizando el criterio definido en `valid_program` y luego se interpreta el estado inicial del programa p.

```
definition execute :: "program ⇒ state option" where
  "execute p ≡ do {
    assert (valid_program p);
    interp (proc_table_of p) (initial_state p)
  }"
```

FIGURA 3.16: Definición de un interpretador para Chloe

```
definition "yields ≡ lambda cs cs'. Some cs →* cs' /wedge is_term cs'"

definition "terminates ≡ lambda cs. ∃ cs'. yields cs cs'"
```

FIGURA 3.17: Definiciones sobre ejecución de programas

3.8.3. Correctitud

Por último, se debe demostrar que el interpretador es correcto. En la figura 3.17 se encuentran dos definiciones con respecto a la ejecución. La primera establece que una ejecución de un estado *cs* produce (`yields`) *cs′* si se pueden tomar pasos cortos desde *cs* hasta *cs′* y *cs′* es un estado final. En segundo lugar, se dice que la ejecución de un estado *termina* si existe algún estado *cs′* tal que sea producido por la ejecución del estado *cs*.

Antes de demostrar el lema de correctitud para el interpretador, se debe demostrar que la semántica de pasos cortos preserva un estado erróneo `None` si tal es alcanzado por el camino de pasos tomados. Tras una ejecución errónea, la misma se queda atascada en un estado `None`.

Lema 3.17 (None_star_preserved)**.**
`None` $\rightarrow * z \longleftrightarrow z =$ `None`

Finalmente se tiene la propiedad de correctitud para el interpretador. El teorema 3.18 establece que si la ejecución de *cs* termina, entonces esa ejecución produce *cs′* si y solo si *cs′* es el resultado que se obtiene de ejecutar el programa en el interpretador.

Teorema 3.18 (interp_correct)**.**
`terminates` *cs* $\Longrightarrow$ (`yields` *cs cs′*) $\longleftrightarrow$ (*cs′* = `interp proc_table` *cs*)

Capítulo 4

Pretty Printer

En este capítulo se describe el proceso de *pretty printing* (traducción) que ocurre en la semántica que permite exportar código C. Se utiliza la implementación de la clase Show de Haskell en Isabelle/HOL hecha por Sternagel y Thiemann (2014) como ayuda en el proceso de traducción. Sternagel y Thiemann implementan un *type class* para funciones `to-string` así como instanciaciones para los tipos estándar de Isabelle/HOL. Ademas permiten la derivación de funciones show para tipos de datos definidos por el usuario. Se instancia esta clase al crear una función `show` para cada uno de los tipos de datos que se definen progresivamente hasta que se torna posible imprimir un programa. En el apéndice E se explica en detalle las cadenas de caracteres concretas que se obtienen como resultado de la traducción. Esta sección se limitará a comentar los aspectos más relevantes del proceso de traducción. Las siguientes secciones explicarán en detalle las cadenas de caracteres concretos que se obtienen como resultado de la traducción.

4.1. Expresiones

Para imprimir expresiones se deben poder imprimir operaciones unarias y binarias (las cuales se encuentran detalladas en el apéndice E). También se debe poder imprimir valores a los que se les ha hecho *casting* desde y hacia apuntadores. Se utiliza el tipo `intptr_t` de C, el cual es lo suficientemente grande como para guardar el valor de un entero así como el de un apuntador. Esto se hace debido a que solo se tienen valores enteros en el lenguaje y las direcciones y los valores enteros son disjuntos al nivel de semántica.

En el nivel de lenguaje C se le debe poder indicar al compilador cuando un valor debe ser interpretado como una dirección y hacerle *cast* como tal. Se permite este *casting* entre

direcciones y enteros durante el proceso de traducción dado que se conoce con seguridad cuando un valor debe ser interpretado como una dirección y cuando debe ser interpretado como un entero, mientras que C no.

Casts Dado que los valores con los que se trabaja deben ser interpretados en C algunas veces como enteros y otras como apuntadores, se debe poder imprimir un *cast* explícito entre esos dos tipos en el programa generado. Se incluyen *casts* a apuntadores cuando se trata con operaciones de referencia, desreferencia y acceso a arreglos. Se quiere hacer *casting* a entero en el caso de asignación de memoria. Una asignación de memoria retorna un apuntador, pero para poder asignarlo a una variable se le debe hacer *casting* como un entero. Dado que todas las variables del programa se declaran con el tipo `intptr_t` y no `intptr_t *`, se debe hacer esto, además se sabe que al trabajar con direcciones las mismas serán interpretadas de la manera correcta ya que se agregan *casts* a apuntadores.

Un *cast* a una dirección se imprime de la siguiente manera: (`intptr_t` $*$) $<$ *expresion* $>$. Un *cast* a un valor entero se imprime de la siguiente manera: (`intptr_t`) $<$ *expresion* $>$. En la tabla 4.1 se encuentran ejemplos del *pretty printing* de *casts*.

Sintaxis abstracta	**Representación en cadena de caracteres**
`Deref` (`V` *foo*)	$*$((`intptr_t` $*$) foo)
`Ref` (`V` *foo*)	((`intptr_t` $*$) &(foo))
`New` (`Const` 9)	(`intptr_t`) `__MALLOC(sizeof(intptr_t)` $*$ (9))

Tabla 4.1: Ejemplos de *pretty printing* para *casts*

Asignaciones de memoria Como fue mencionado en el capítulo 3 el comportamiento de la función de asignación de este trabajo y la función de asignación de memoria de C difieren debido a que en este trabajo se supone que la memoria es ilimitada. Por lo tanto no se puede simplemente traducir la asignación de memoria a una llamada a la función `malloc` en C.

Se debe envolver la llamada a la función `malloc` de C en otra función que aborte el programa en el caso en que exista una falla por falta de memoria. Se define una función "`__MALLOC`" que toma el tamaño del nuevo bloque de memoria que será asignado con `malloc`, hace la llamada a `malloc` y retorna el apuntador al nuevo bloque de memoria en el caso de que la llamada sea exitosa y si la llamada falla entonces aborta el programa con el código de salida 3. Se define el código de salida 3 como un código de salida erróneo que significa que hubo un error al asignar memoria para poder detectar este error luego en el proceso de pruebas.

4.2. Programas

En esta sección se discute la traducción de un programa completo.

Archivos de cabecera y chequeo de límites En el código C que será exportado se quiere incluir los archivos de cabecera para las librerías estándar de C (`stdlib.h` y `stdio.h`). Además se incluyen dos archivos de cabecera adicionales, una permite el uso del tipo `intptr_t` y la otra es donde se encuentran los macros con las definiciones de los límites de los tipos enteros. Estas son `limits.h` y `stdint.h`, respectivamente. También se desea incluir el archivo de cabecera que define ciertos macros utilizados por el arnés de pruebas con fines de pruebas, este tema será tratado con más detalle en el capítulo 5. Por último, se incluye el archivo de cabecera que contiene la función para manejar las llamadas a `malloc`. Antes de generar alguna otra parte del programa se quiere imprimir las directivas para incluir los archivos de cabecera mencionados anteriormente.

Además, el proceso de traducción genera un programa que será compilable y ejecutable. Esto es cierto para las arquitecturas que soporten al menos el mismo rango para los tipos enteros que esta abstracción. Esto significa que, cualquier arquitectura donde el programa sea compilado y ejecutado debe cumplir con las restricciones supuestas por esta semántica para valores enteros. Por lo tanto, se utiliza el preprocesador de C para imprimir una verificación de límites de enteros que afirma que los macros que contienen los límites superiores e inferiores para el tipo entero se encuentren definidos. En este caso, se debe verificar que los macros `INTPTR_MIN` y `INTPTR_MAX` estén ambos definidos. A continuación se verifica que los límites definidos en tales macros son iguales a los límites supuestos en la semántica (los límites supuestos son `INT_MIN` y `INT_MAX` y se definen en la figura 3.2).

El tipo utilizado para la traducción al igual que la precisión de los enteros puede ser cambiado ya que están parametrizados. Se definen los valores y se utilizan en la semántica y en el proceso de traducción. Las definiciones para el tipo utilizado en esta traducción son las siguientes:

```
definition "dflt_type ≡ ''intptr_t''"
definition "dflt_type_bound_name ≡ ''INTPTR''"

definition "dflt_type_min_bound_name ≡ dflt_type_bound_name @ ''_MIN''"
definition "dflt_type_max_bound_name ≡ dflt_type_bound_name @ ''_MAX''"
```

donde @ representa la operación *append* entre cadenas de caracteres. Nótese que la precisión del tipo utilizado como `dflt_type` debe corresponder a la precisión definida en `int_width`.

Variables globales La impresión de variables globales se hace de la misma manera que para las variables locales, siguiendo el siguiente formato:

$$\texttt{intptr\_t} < variable\_name_0 >;$$
$$\vdots$$
$$\texttt{intptr\_t} < variable\_name_n >;$$

Programas Para imprimir un programa se imprimen las directivas `include` para los archivos de cabecera, luego la verificación de los límites para los enteros, las declaraciones de las variables globales y, por último, se imprime cada función en el programa separado por un carácter de salto de línea. En el apéndice J se encuentra la definición de un programa factorial en Isabelle y el código C generado para el mismo.

4.3. Exportando código C

Ahora que hay una manera de traducir un programa completo desde la semántica a código C, se puede exportar el código generado para C.

Solo se exporta código para programas válidos. Para garantizar esto se define una función que prepara un programa para su exportación:

```
definition prepare_export :: "program ⇒ string option" where
  "prepare_export prog ≡ do {
    assert (valid_program prog);
    Some (shows_prog prog '''')
  }"
```

Esta función toma un programa y verifica que sea válido, de acuerdo a la definición de validez dada en 3.4. Si no es válido, se retorna un valor `None`, en caso contrario se genera la cadena de caracteres que contiene el programa en C con la función `shows_prog`. [1]

Se exporta código C a archivos externos. El nombre del archivo está dado por el nombre del programa en su definición. Se define una función en ML dentro de Isabelle que exporta el código C de un programa. La definición de la misma se encuentra en el apéndice F.

Esta función toma cuatro parámetros. El primer parámetro de la función `export_c_code` es un `string option` esto corresponde a la representación en cadena de caracteres del código

[1] No se incluye el código para las funciones show en este documento sino que se explica como funcionan. Los archivos de Isabelle pueden ser chequeados para detalles de implementación.

C generado por el programa. Si recibe un valor `None` significa que la función `prepare_export` falló y no se debe generar un programa en C. El segundo parámetro es la ruta al directorio al cual se quiere exportar el programa, este parámetro es una ruta relativa al directorio donde se encuentra la teoría de Isabelle que contiene las directivas para hacer *pretty printing*. El tercer parámetro es el nombre del programa. El último parámetro es el contexto de la teoría actual.

Si la ruta dada es una cadena vacía el código generado se imprimirá a la vista de salida de Isabelle. Esta función crea un nuevo archivo con el nombre indicado en los parámetros con "`.c`" agregado al final (es decir, $<$ *nombre* $>$ `.c`) en el directorio indicado por el parámetro de la ruta. El usuario puede entonces encontrar el código generado en el directorio indicado.

También se define una función en ML que se utiliza para el caso donde se escribe un programa erróneo a propósito y se espera que su ejecución y traducción a C fallen:

```
fun expect_failed_export (SOME _) = error ``Expected failed export''
  | expect_failed_export NONE = ()
```

La función `expect_failed_export` genera un error en Isabelle si se genera código a partir del programa para el cual se esperaba una falla y no realizará acción alguna cuando no se genere código. Tener este tipo de función será útil más adelante para la realización de pruebas.

Capítulo 5

Proceso de Pruebas

En los capítulos 3 y 4, se formalizó la semántica para Chloe y se describió el proceso de traducción a código C. Ahora se describe el proceso de pruebas hecho para incrementar la confianza en el proceso de traducción. Se garantiza que el código generado de la semántica o bien termina con un error por falta de memoria o produce un resultado igual que el del mismo programa ejecutado en el ambiente Isabelle/HOL. Se dice que los resultados de ejecutar el programa dentro de Isabelle y el programa generado en la máquina son iguales si los estados finales producidos por ambos son equivalentes. Esto significa que se calcula el estado final producido por la ejecución de la semántica y se verifica que luego de ejecutar el programa generado el estado es equivalente, es decir, los contenidos de la memoria accesible y las variables globales son iguales. Al generar programas se puede simplemente genera el código C por si solo o se puede incluir un conjunto de pruebas extras en forma de macros de C que garantizan la condición de equivalencia mencionada anteriormente.

En las siguientes secciones se describen los arneses de pruebas utilizados para generar las pruebas para el código. Además se da una descripción mas detallada del significado que tiene que dos estados finales sean equivalentes. Por último se habla un conjunto de pruebas y programas de ejemplos escritos en la semántica. Sólo se genera código para programas válidos; esto es, se irá a un estado erróneo si surge algún comportamiento indefinido. En este conjunto de pruebas se incluyen programas que se espera que alcancen un estado erróneo ya que presentan comportamiento indefinido o casos bordes. Para estos programas *incorrectos* no se generará código C. También se presentan programas de ejemplo, tales como algoritmos de ordenamiento, para demostrar como funciona la semántica y el proceso de generación de código.

5.1. Equivalencia de estados finales

Se considera que un estado final producido por la ejecución de la semántica es igual al estado final de su programa generado en C cuando, al final de la ejecución, los valores de las variables globales son iguales para ambos casos y cada bloque de memoria accesible tiene contenidos iguales. Se describe a continuación como se verifica esta equivalencia entre estados finales mediante el uso de pruebas.

5.1.1. Generación de pruebas

Se pueden generar pruebas para los programas. Estas serán ejecutadas al final de la ejecución del programa entero y verificarán que el estado final del programa generado es el mismo que el estado final de la ejecución de la semántica. Para comparar estos estados finales se comparan los valores de las variables globales al final de la ejecución con aquellos en el estado final de la ejecución de la semántica.

La dirección en la que las pruebas se hacen es tomando los valores del estado final de la ejecución de la semántica y verificando que la ejecución del código generado tenga los mismos valores que son esperados de acuerdo a la ejecución de la semántica.

Se introducen a continuación los distintos tipos de pruebas que se hacen dependiendo del contenido de una variable global.

Valores enteros

Cuando el contenido de una variable global es un valor entero, simplemente se debe generar una prueba que compruebe que el valor entero en la variable global al final de la ejecución del código en C es el mismo que se obtiene de la valuación de la variable global en el estado final de la ejecución de la semántica.

Apuntadores

Para verificar los apuntadores se tienen dos casos. El caso del apuntador a `NULL` y el caso donde el apuntador no es nulo.

Para valores `NULL` se genera una prueba, similar a aquella para valores enteros, donde se comprueba que el contenido de la variable global es `NULL`.

En el caso de apuntadores que tengan valor diferente a `NULL`, se tiene un apuntador al bloque en memoria y se quiere verificar que el contenido del bloque en memoria, al final de la

ejecución del programa generado, es igual al contenido del mismo bloque en el estado final en la semántica. El bloque completo califica como memoria accesible, por lo que se debe verificar el contenido de cada celda en el bloque. Para cada celda en el bloque se generan chequeos dependiendo de cual es el contenido que se espera que tenga la celda en memoria, es decir, un valor entero, un valor `NULL` o un apuntador.

En el caso de chequeos para valores del tipo apuntador, se sigue cada apuntador hasta que se alcanza o bien un valor entero o un apuntador que ya fue seguido. Al alcanzar un valor entero, se genera una prueba del tipo entero y al alcanzar un apuntador que ya fue seguido, se sabe que el camino no contiene apuntador alguno a memoria inválida. Se verifica que la dirección del inicio del bloque es la misma que se obtiene al ajustar el apuntador que se siguió al inicio del bloque. Para hacer esto, se deben seguir los apuntadores en un cierto orden y se debe mantener un conjunto de bloques de memoria ya *visitados*. De esta manera, al encontrar un apuntador a un bloque de memoria que ya fue chequeado (o *visitado*) se puede parar y comparar las direcciones en lugar de seguir los apuntadores de manera cíclica indefinidamente.

Se presenta aquí la idea intuitiva detrás de la generación de pruebas y en las siguientes secciones se describen los detalles de implementación para los arneses de pruebas, ambos en Isabelle y C.

5.2. Arnés de prueba en Isabelle/HOL

Primero, se define un nuevo tipo de datos por cada tipo de instrucción que se puede generar. Esta definición se encuentra en la figura 5.1.

Se tienen cuatro tipos de instrucciones de prueba diferentes que se pueden generar:

- `Discover` representa la instrucción que agrega un bloque a la lista de bloques *visitados*. El `string` es la representación en cadena de caracteres de la expresión en C y el `nat` representa el número de identificación del bloque de memoria actual. Las direcciones reales para los bloques de memoria asignada cambiarán con cada ejecución del programa en la máquina. La instrucción `discover` empareja la dirección real para el inicio de un bloque con el número de bloque correspondiente en la representación abstracta. Para ello se generan variables locales con la función `base_var_name` llamadas `__test_harness_x_`n donde n es el número de identificación del bloque y en esas variables se guarda la dirección real del inicio del bloque para una ejecución en particular.

```
datatype test_instr =
  Discover string nat
| Assert_Eq string int_val
| Assert_Eq_Null string
| Assert_Eq_Ptr string nat

fun adjust_addr :: "int ⇒ string ⇒ string"
  where
  "adjust_addr ofs ca = shows_binop (shows ca) (''-'') (shows ofs) ''''"

definition ofs_addr :: "int ⇒ string ⇒ string"
  where
  "ofs_addr ofs ca =
    (shows ''*'' o
      shows_paren (shows_binop (shows ca) (''+'') (shows ofs))) ''''"

definition base_var_name :: "nat ⇒ string" where
  "base_var_name i ≡ ''__test_harness_x_'' @ show i"
```

FIGURA 5.1: Definiciones del arnés de prueba en Isabelle

- `Assert_Eq` representa la instrucción que chequea que el valor de una expresión sea el mismo valor entero que se espera que tenga. El `string` es la representación en cadena de caracteres de la expresión en C y el `int_val` representa el valor que se espera que esa expresión tenga de acuerdo al estado final en la ejecución de la semántica.

- `Assert_Eq_Null` representa la instrucción que chequea que el valor de una expresión sea `NULL`. El `string` es la representación en cadena de caracteres de la expresión en C.

- `Assert_Eq_Pointer` representa la instrucción que chequea que el valor de un apuntador apunte al mismo bloque al que se espera que apunte. El `string` es la representación en cadena de caracteres de la expresión en C y el `nat` representa el número de identificación del bloque de memoria.

Se tienen también ciertas funciones auxiliares que asisten en el proceso de generación de pruebas. La función `adjust_addr` toma un desplazamiento y una representación en cadena de caracteres de una expresión en C (cuya evaluación resulta en un apuntador) y produce una representación en cadena de caracteres que ajusta la dirección al inicio del bloque al sustraer el desplazamiento de la misma. La función `ofs_addr` toma un desplazamiento y la representación en cadena de caracteres de una expresión en C (cuya evaluación resulta en un apuntador) y produce una representación en cadena de caracteres que ajusta la dirección para que apunte a la celda especificada por el desplazamiento al sumarle el mismo a la dirección. Por último, la función `base_var_name` dado un número natural n produce la variable que se utiliza para el proceso de pruebas, la cual guarda la dirección que apunta al inicio del bloque

n. Esta variable siempre tendrá el prefijo `__test_harness_x_` mas el número n. Por ejemplo, para el número de bloque 2 la función produce: "`__test_harness_x_2`".

Anteriormente se mencionó que los apuntadores deben ser seguidos hasta que se alcance o bien un entero o un valor `NULL` o hasta que se alcance un apuntador que ya haya sido seguido. Para hacer esto se deben seguir los apuntadores en un orden determinado y se deben mantener un conjunto de bloques ya *visitados*. De esta manera al encontrar un apuntadora a un bloque de memoria que ya fue chequeado (o *visitado*) se puede parar y comparar los apuntadores en lugar de seguir los apuntadores de manera cíclica indefinidamente, chequeando una y otra vez partes de la memoria que ya fueron revisadas. Los apuntadores se siguen en el orden dado por el algoritmo de búsqueda en profundidad (DFS) [1]. En el apéndice G se presenta el algoritmo utilizado para seguir un apuntador. El algoritmo toma un conjunto de números naturales que son los bloques que ya han sido visitados, una dirección (la que se está siguiendo) y una representación en cadena de caracteres de la expresión en C (que debería contener la misma dirección). Produce un nuevo conjunto de bloques visitados y una lista de instrucciones de prueba que se deben generar.

El algoritmo opera de la siguiente manera: Primero, se intenta acceder al bloque, para ver si la memoria está libre o tiene contenido alguno. Si la memoria está libre se retorna el mismo conjunto de bloques visitados y no se genera instrucción extra alguna. Si la memoria no está libre, se ajusta la dirección al inicio del bloque, esto es porque al chequear la memoria queremos revisar los bloques completos ya que son parte de la memoria alcanzable. Si el bloque que se está revisando pertenece ya al conjunto de visitados, simplemente se retorna el mismo conjunto de visitados y una instrucción `Assert_Eq_Ptr` para chequear los apuntadores. No obstante, si el bloque que se está revisando es un bloque que no ha sido *visto* aun, se agrega al conjunto de visitados y se agrega una instrucción `Discover` a la lista de instrucciones generadas.

Luego se chequea el contenido del bloque de memoria, comenzando por la primera celda hasta la última celda del bloque. Al chequear cada celda se revisa si el contenido de la celda es un valor entero, `NULL` o una dirección. Si la celda contiene un valor entero, se retorna el mismo conjunto de visitados y se agrega una instrucción `Assert_Eq` a la lista de instrucciones generadas. Si la celda contiene un valor `NULL`, se retorna el mismo conjunto de visitados y se agrega una instrucción `Assert_Eq_Null` a la lista de instrucciones generadas. Finalmente, si la celda contiene una dirección, se debe seguir esa dirección haciendo una llamada recursiva a `dfs` antes de continuar revisando el bloque de memoria actual. Al retornar de esta llamada,

[1] Por sus siglas en inglés *Depth-First Search*

se retorna el nuevo conjunto de visitados resultante de la llamada recursiva y se agrega a la lista de instrucciones generadas hasta el momento la lista de instrucciones que retorna la llamada recursiva.

5.3. Arnés de prueba en C

Con el fin de apoyar el proceso de pruebas en C se deben tener algunos macros que corresponden a aquellas instrucciones generadas en Isabelle. Se crea un archivo de cabecera en C donde se definen los macros necesarios para realizar las pruebas así como algunas variables útiles. Este archivo de cabecera se presenta en el apéndice H. Allí se pueden encontrar las definiciones para los macros correspondientes a las instrucciones `Discover`, `Assert_Eq`, `Assert_Eq_Null` y `Assert_Eq_Ptr`. Para guardar el conjunto de bloques visitados se utiliza un *hash set*. La implementación del *hash set* fue hecha por Sergey Avseyev y se encuentra en línea (Avseyev, 2013). Se definen variables que contienen el número total de pruebas, el número de pruebas exitosas y fallidas y el *hash set* de bloques visitados.

Las instrucciones de pruebas son traducidas mediante *pretty printing* a macros de C utilizando la función `test_instructions`:

```
definition tests_instructions :: "test_instr list ⇒ nat ⇒ shows" where
  "tests_instructions l ind ≡ foldr (λ
      (Discover ca i) ⇒
        indent_basic ind
          (shows ''__TEST_HARNESS_DISCOVER '' o
            shows_paren
             ( shows ca o shows '', '' o shows (base_var_name i)))
    | (Assert_Eq ca v) ⇒
        indent_basic ind
          (shows ''__TEST_HARNESS_ASSERT_EQ '' o
            shows_paren ( shows ca o shows '', '' o shows v))
    | (Assert_Eq_Null ca) ⇒
        indent_basic ind
          (shows ''__TEST_HARNESS_ASSERT_EQ_NULL '' o
            shows_paren ( shows ca ))
    | (Assert_Eq_Ptr ca i) ⇒
        indent_basic ind
          (shows ''__TEST_HARNESS_ASSERT_EQ_PTR '' o
            shows_paren
             ( shows ca o shows '', '' o shows (base_var_name i)))
    ) l"
```

Para cada bloque para el cual se genera una instrucción `Discover` se debe también imprimir (*pretty print*) una declaración para cada una de las variables utilizadas en las pruebas. Esto se hace de la siguiente manera:

```
  definition tests_variables :: "test_instr list ⇒ nat ⇒ shows" where
    "tests_variables l ind ≡ foldr (λ
      (Discover _ i) ⇒
        indent_basic ind
          (shows dflt_type o shows '' *'' o shows (base_var_name i))
      | _ ⇒ id
      ) l"
```

Por último, se puede obtener la lista de instrucciones que deben ser generadas usando la función `emit_global_tests`. Dada una lista de variables, genera una lista de instrucciones de prueba para las cuales se genera código en C. Esta función se define como sigue:

```
definition
  emit_globals_tests ::
    "vname list ⇒ state ⇀ (nat set × test_instr list)"
where "emit_globals_tests ≡ λvnames (σ,γ,μ).
  fold_option (λx (D,emit). do {
    case γ x of
      Some vo ⇒ do {
        let cai = x;
        case vo of
            None ⇒ Some (D,emit)
          | Some (I v) ⇒ Some (D,emit @ [Assert_Eq cai v])
          | Some (NullVal) ⇒ Some (D,emit @ [Assert_Eq_Null cai] )
          | Some (A addr) ⇒ do {
              (D,emit ← dfs μ D addr cai;
              Some (D,emit@emit')
            }
      }
    | _ ⇒ Some (D,emit)
  }
  ) vnames ({},[])"
```

5.4. Pruebas

5.4.1. Generación de código con pruebas

En la sección 4.3 se describe una forma de exportar programas en C. Existe una segunda manera de exportar programas en C donde además se genera código C para verificar la igualdad de los estados finales.

```
definition prepare_test_export :: "program ⇒ (string × string) option"
where "prepare_test_export prog ≡ do {
  code ← prepare_export prog;
  s ← execute prog;
  let vnames = program.globals prog;
  (_,tests) ← emit_globals_tests vnames s;
  let vars = tests_variables tests 1 ''';
  let instrs = tests_instructions tests 1 ''';
  let failed_check = failed_check prog;
  let init_hash = init_disc;
  let nl = ''⇓'';
  let test_code =
    nl @ vars @ nl @ init_hash @ nl @ instrs @ nl @ failed_check @ nl @ ''}'';
  Some (code,test_code)
}"
```

FIGURA 5.2: Función que prepara un programa para exportación con pruebas

Anteriormente, se definió la manera en que cada construcción necesaria para las pruebas se imprime (*pretty printing*), ahora se describe como se genera el código con las pruebas.

Se tiene una función similar a `prepare_export` que prepara un programa para ser exportado con pruebas generadas, se define en la figura 5.2 (donde ⇓ representa el carácter de salto de línea). Primero, se obtiene el código del programa sin pruebas utilizando la función `prepare_export`. Solo se generan las pruebas para programas válidos cuya ejecución produzca un estado final, por lo tanto se debe verificar esto mediante la ejecución del programa. Luego, se crea una lista de pruebas para las variables globales. Más adelante, se genera la cadena de caracteres para las declaraciones de variables, una cadena de caracteres para inicializar el *hash set* y la cadena de caracteres para las llamadas a los macros definidos en C. Se tienen tres variables en el arnés de pruebas en C que llevan la cuenta de cuantas pruebas son ejecutadas, cuantas fallan y cuantas son exitosas. Se quiere obtener algún tipo de información sobre el resultado de correr las pruebas generadas para el código. Para ello se genera código que imprime a la salida estándar (luego de ejecutar el programa) los resultados del proceso de pruebas, es decir, el número de pruebas exitosas y fallidas. Por último, `prepare_test_export` produce una tupla que contiene el código para el programa sin pruebas y las pruebas generadas para el mismo.

Con el fin de exportar el código con las pruebas agregadas, se define una nueva función en ML llamada `generate_c_test_code`, su definición se encuentra en el apéndice K. Esta función toma cuatro parámetros. El primer parámetro de la función es de tipo `(string, string) option` que corresponde a la tupla que contiene la representación en cadena de caracteres del programa en código C y las pruebas para ese programa, respectivamente. Si la función recibe un valor `None`, significa que la función `prepare_test_export` falló y no se debe generar un

programa en C. El segundo parámetro es la ruta al directorio al que se quiere exportar el programa. Este parámetro es una ruta relativa al directorio donde se encuentra la teoría de Isabelle que contiene las directivas para hacer *pretty printing*. El tercer parámetro es el nombre del programa. El último parámetro es el contexto de la teoría actual.

Si la ruta dada es una cadena de caracteres vacía el código generado se imprimirá a la vista de salida de Isabelle. Esta función crea un nuevo archivo con el nombre indicado en los parámetros con "`.c`" agregado al final (es decir, $<$ *nombre* $>$ `.c`) en el directorio indicado por el parámetro de la ruta. El usuario puede entonces encontrar el código generado en el directorio indicado.

Esta función escribe el código C para un programa en un archivo y le agrega las pruebas generadas para el mismo al final de la función `main`.

La función `expect_failed_test` es muy similar a la función `expect_failed_export` presentada en la sección 4.3 pero con un mensaje de error diferente. Esta función genera un error en Isabelle si se genera código a partir del programa para el cual se esperaba una falla y no realizará acción alguna cuando no se genere código.

5.4.2. Pruebas incorrectas

Al desarrollar una *suite* de pruebas es importante considerar casos incorrectos. Se escribió un conjunto de programas para los cuales se espera que la generación de código falle. Al utilizar las funciones `expect_failed_export` y `expect_failed_test` definidas en las secciones 4.3 y 5.4.1, respectivamente, se pueden escribir programas incorrectos y al generar código C para ellos se le puede indicar a Isabelle que debe esperar que esos procesos fallen de modo que no genere un error.

Tener programas incorrectos es muy útil ya que sirven como pruebas de regresión. Al agregar nuevas características a la semántica se pueden correr todas las pruebas en la *suite* de pruebas. Si alguno de esos programas incorrectos se ejecuta de manera exitosa en la semántica y se genera código en C, se genera un error en Isabelle que indica que se está generando código C para programas que se espera que sean fallidos. Estas pruebas son útiles para detectar errores si los cambios hechos alteran la semántica existente.

Sin embargo, es importante notar que para que una *suite* de pruebas de regresión sea útil debe cubrir tantos casos como sea posible, lo cual, al trabajar con un lenguaje mas grande que Chloe, requiere una cantidad considerable de pruebas escritas.

5.5. Programas de ejemplo

Además de las pruebas descritas en esta sección se presenta un conjunto de programas de ejemplo en Chloe. Estos tienen la finalidad de mostrar como se escriben programas en Chloe y como funcionan la ejecución y la generación de código. Estos se encuentran en el código fuente presentado junto a este trabajo.

5.6. Ejecución de pruebas

5.6.1. Ejecución de pruebas en Isabelle

En el código fuente presentado con este trabajo se tiene un directorio que contiene todas las pruebas y programas de ejemplo escritos para Chloe. Es necesaria una forma de ejecutar esas pruebas en Isabelle automáticamente.

En el código fuente se incluye una teoría de Isabelle llamada "`All_Tests.thy`" la cual es simplemente un archivo de Isabelle que importa cada prueba escrita para Isabelle, ambas correctas e incorrectas. Al abrir este archivo en Isabelle, todos los archivos correspondientes a las pruebas y programas de ejemplo serán cargados. Se tendrán entonces dos casos para cada prueba.

Para pruebas correctas y programas de ejemplos, se generará código. Para pruebas incorrectas, no se generará código. En el caso en el que ocurre un error, bien sea que se genere código para pruebas inválidas, o que el código no se genere para pruebas válidas, habrá un error. Es posible ver estos de manera fácil en la vista llamada 'Theories' de la interfaz gráfica de Isabelle ya que los archivos con un error son marcados en rojo.

5.6.2. Ejecución de pruebas en C

Cuando se genera código exitosamente, se desea compilar y correr las pruebas de manera automática. Se tiene un *Makefile* que compila cada prueba en la *suite* así como también un *shell script* de *bash* que corre cada prueba en la *suite*. El resultado de ejecutar las pruebas será el número de pruebas exitosas y/o fallidas. Cuando una prueba falla, el resultado se imprime en color rojo para hacerlo mas visible al usuario. Además, otros comportamientos son atrapados por el *script* y mostrados al usuario, tales como errores por falta de memoria, *segmentation faults* y programas que terminen con alguno de los códigos de salida reservados (Cooper, 2014). Este *script* se presenta en el apéndice I.

5.7. Resultados

Tener una *suite* de pruebas que permite verificar si el proceso de traducción es hecho correctamente fue muy valioso durante el desarrollo de este trabajo. Originalmente, se tenían programas simples escritos en el lenguaje que ayudaban a verificar intuitivamente que el proceso de traducción se estaba haciendo de manera correcta y que la semántica del programa no estaba siendo cambiada. Posteriormente, fue necesario automatizar el proceso de pruebas y diseñar pruebas más específicas. Se procedió a agregar el arnés de pruebas y a crear la batería de pruebas. Los resultados obtenidos de las pruebas fueron positivos. Todos los casos incorrectos fallaron, como era esperado. Para todos los casos correctos se genera código con pruebas y todas las pruebas generadas para estos programas son ejecutadas de manera exitosa.

Es importante notar que los casos de prueba fueron escritos por los mismos desarrolladores del proyecto, por lo que no se pueden establecer métricas exactas acerca de los resultados obtenidos durante el proceso de pruebas. Es posible que algunos casos no estén contemplados ni cubiertos en la *suite* de pruebas dado que las pruebas fueron escritas por los desarrolladores. Es por ello que nuevos casos de prueba pueden, y deberían, ser agregados a la *suite* de pruebas con el fin de continuar incrementando la confianza en el proceso de traducción. Por otra parte, nuevas pruebas deben ser agregadas al agregar nueva funcionalidad a la semántica.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este trabajo se logró formalizar exitosamente la semántica para un lenguaje imperativo llamado Chloe, el cual abarca un subconjunto del lenguaje C. Chloe tiene las siguientes características: variables, arreglos, aritmética de apuntadores, ciclos, condicionales, funciones y memoria dinámica. Se formalizó una semántica de pasos cortos en el probador de teoremas Isabelle/HOL para Chloe y se demostró que la misma es determinística. Además, se presenta un interpretador para el lenguaje, lo que permite que los programas escritos en Chloe sean ejecutados en el entorno de Isabelle/HOL. Para escribir el interpretador era necesario demostrar que la semántica era determinística.

Otro resultado de este trabajo fue la presentación de un generador de código para el lenguaje Chloe. Este generador de código traduce programas de la semántica formal a código C real que puede ser ejecutado. Se garantiza que el programa generado puede ser compilado y ejecutado en una máquina si no posee comportamiento indefinido, cumple con las restricciones supuestas en la formalización de la semántica (sección 3.5) y no se queda sin memoria durante la ejecución. Además, solo se genera código en C para programas que sean válidos y se ejecuten correctamente en el interpretador.

Al formalizar la semántica se presentaron varios problemas, tales como sólo permitir que programas con comportamiento definido fueran ejecutados en la semántica. Esto significa que se tuvo que detectar comportamiento indefinido, de acuerdo al estándar de C (ISO/IEC, 2007), tales como el *overflow* de enteros y considerarlo como un error en la semántica. Otro problema se presentó al formalizar la semántica para la función de asignación de memoria ya que se supone que una cantidad ilimitada de memoria está disponible, mientras que los

recursos en una máquina son limitados. Para solventar esta problemática, se decidió cambiar la manera en que se traducía una llamada a la función de asignación de memoria a código C. Se rodea la función `malloc` de C en una nueva definida por el usuario que aborta el programa si una llamada a `malloc` falla. También se supone una restricción sobre la arquitectura de la máquina destino donde el código será ejecutado y se genera código que verifica que la máquina satisface las suposiciones.

Por otra parte, se presenta un arnés de pruebas y una *suite* de pruebas para el proceso de traducción. La finalidad de esta *suite* de pruebas es incrementar la confianza en el proceso de traducción y asegurarse que la semántica de un programa en Chloe no cambia al ser traducido a código C. En la *suite* de pruebas se incluyen casos incorrectos que cubren los casos bordes o casos donde no se debería generar código C ya que el programa encuentra un error. Además, se incluyen programas de ejemplo para demostrar como funciona el lenguaje y como se traducen los programas a código C. Existe también un conjunto de pruebas correctas para las que se espera generar código. La batería completa de pruebas puede ser encontrada en el código fuente presentado con este trabajo.

Para garantizar que la semántica de un programa no es cambiada por el proceso de traducción, se incluye una manera de generar código con chequeos. Estos chequeos tienen la finalidad de verificar que el estado final de un programa ejecutado en el interpretador dentro de Isabelle/HOL es equivalente al estado final del programa ejecutado y compilado. Dos estados son considerados equivalentes cuando las variables globales y la memoria dinámica accesible, al final de la ejecución, tienen el mismo contenido. Para ello, se escribe un arnés de pruebas en Isabelle que traduce las pruebas que se deben hacer a un conjunto de macros en C que se definen fuera de Isabelle/HOL, y se incluyen en el proceso de compilación, que verifican la equivalencia entre estados finales. Con el fin de no correr estas pruebas manualmente se crea un *bash script* que corre todas las pruebas existentes automáticamente.

6.2. Trabajo futuro

En esta sección se señalan algunas direcciones que puede tomar este trabajo en un futuro. El marco de tiempo disponible para hacer este trabajo hizo imposible incluir estas características en el alcance del mismo.

En primer lugar, se puede mejorar el proceso de pruebas al reconocer lexicográficamente pruebas de C de una *suite* de pruebas externa y traducirlas a Chloe. Esto permitiría ofrecer métricas mas exactas para los resultados del proceso de pruebas, por ejemplo, que tanto de

la *suite* de pruebas abarca exitosamente el trabajo. Un ejemplo de esto sería tomar una *suite* de pruebas hecha para compiladores de C, tales como las *suites* de prueba de gcc (Stallman and the GCC Developer Community, 2015) y reducir las pruebas a aquellas que puedan ser traducidas a la semántica, traducirlas de C a Chloe y generar código con chequeos a partir de las mismas. Luego de traducir a Chloe se puede generar código con chequeos para las mismas con el fin de mejorar la *suite* de pruebas que se presenta con este trabajo.

Otra dirección interesante que se podría tomar es la de formalizar una semántica axiomática con el fin de razonar sobre programas y sus propiedades en Chloe. Dado que Chloe posee apuntadores como una de sus características, sería necesario formalizar una lógica de separación (Reynolds, 2002) para poder razonar sobre apuntadores en los programas. Al extender el trabajo en esta dirección sería posible demostrar propiedades de correctitud parcial y total para los programas.

Una de las características que no se encuentran incluidas en el alcance del trabajo es un sistema de tipos estático que sea demostrado como correcto y sólido. Esto permitiría razonar sobre seguridad de tipos para los programas escritos en Chloe.

Existe un marco de trabajo llamado Isabelle Refinement Framework (Lammich, 2012), que proporciona una manera de formular algoritmos no determinísticos en un estilo monádico y refinarlos para obtener un algoritmo ejecutable. Proporciona herramientas para razonar sobre estos programas. Otra fuente de trabajo futuro sería vincular este lenguaje al Isabelle Refinement Framework, de modo que los programas del marco de trabajo puedan ser refinados a programas en Chloe.

Por último, el conjunto de características que actualmente son compatibles con Chloe es limitado. Otra forma de mejorar los resultados de este trabajo es aumentar el conjunto de características compatibles con Chloe. Esto puede incluir expandir el conjunto de expresiones e instrucciones (por ejemplo, agregar compatibilidad para structs y unions), agregar operaciones de entrada y salida o agregar soporte para concurrencia.

Bibliografía

Avseyev, S. (2013). Hash set c implementation. `https://github.com/avsej/hashset.c`, [Accessed: 2015-08-07].

Blazy, S. and Leroy, X. (2009). Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288.

Boldo, S., Jourdan, J.-H., Leroy, X., and Melquiond, G. (2013). A formally-verified C compiler supporting floating-point arithmetic. In *ARITH, 21st IEEE International Symposium on Computer Arithmetic*, pages 107–115. IEEE Computer Society Press.

Cooper, M. (2014). *Advanced Bash-Scripting Guide*. `http://www.tldp.org/LDP/abs/html/index.html`.

Greenaway, D., Lim, J., Andronick, J., and Klein, G. (2014). Don't sweat the small stuff: Formal verification of c code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 429–439.

Haftmann, F. (2015). *Code generation from Isabelle/HOL theories*. `http://isabelle.in.tum.de/dist/Isabelle2015/doc/codegen.pdf`.

ISO/IEC (2007). *Programming Languages — C*. ISO/IEC 9899:TC3.

Lammich, P. (2012). Refinement for monadic programs. *Archive of Formal Proofs*. `http://afp.sf.net/entries/Refine_Monadic.shtml`, Formal proof development.

Leroy, X. and Blazy, S. (2008). Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31.

Lochbihler, A. (2013). Native word. *Archive of Formal Proofs*. `http://afp.sf.net/entries/Native_Word.shtml`, Formal proof development.

Nielson, H. R. and Nielson, F. (2007). *Semantics with Applications: An Appetizer*. Springer.

Nipkow, T. and Klein, G. (2014). *Concrete Semantics with Isabelle/HOL*. Springer.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2015). *Isabelle/HOL: A Proof Assistant for Higher-Order-Logic*. `http://isabelle.in.tum.de/dist/Isabelle2015/doc/tutorial.pdf`.

Norrish, M. (1998). *C formalized in HOL*. PhD thesis, University of Cambridge.

Norrish, M. and Slind, K. (2014). *The HOL System Description*.

Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. `http://www.cs.cmu.edu/~jcr/seplogic.pdf`.

Stallman, R. M. and the GCC Developer Community (2015). *GNU Compiler Collection Internals*. Free Software Foundation, Inc. `https://gcc.gnu.org/onlinedocs/gccint.pdf`.

Sternagel, C. and Thiemann, R. (2014). Haskell's show class in isabelle/hol. *Archive of Formal Proofs*. `http://afp.sf.net/entries/Show.shtml`, Formal proof development.

Team, T. C. D. (2015). *The Coq proof assistant reference manual*. TypiCal Project.

Tennent, R. D. (1991). *Semantics of Programming Languages*. Series in Computer Science. Prentice Hall International.

Wenzel, M. (2015). *The Isabelle/Isar Implementation*. `http://isabelle.in.tum.de/dist/Isabelle2015/doc/implementation.pdf`.

Apéndice A

Definiciones para un programa

```
record program =
name :: string
globals :: vname list
procs :: fun_decl list

reserved_keywords =
[''auto'', ''break'', ''case'', ''char'', ''const'', ''continue'',
 ''default'', ''do'', ''double'', ''else'', ''enum'', ''extern'',
 ''float'', ''for'', ''goto'', ''if'', ''inline'', ''int'', ''long'',
 ''register'', ''restrict'', ''return'', ''short'', ''signed'',
 ''sizeof'', ''static'', ''struct'', ''switch'', ''typedef'',
 ''union'', ''unsigned'', ''void'', ''volatile'', ''while'',
 ''_Bool'', ''_Complex'', ''_Imaginary'']

test_keywords =
[''__test_harness_num_tests'', ''__test_harness_passed'',
 ''__test_harness_failed'', ''__test_harness_discovered'' ]

definition valid_program :: program ⇒ bool where
valid_program p ≡
  distinct (program.globals p)
⋀ distinct (map fun_decl.name (program.procs p))
⋀ (∀ fd ∈ set (progarm.procs p). valid_fun_decl fd)
⋀ ( let
     pt = proc_table_of p
   in
     ''main'' ∈ dom pt
     ⋀ fun_decl.params (the (pt ''main'')) = [])
⋀ ( let
     prog_vars = set ((program.globals p) @
       collect_locals (program.procs p));
     proc_names = set (map (fun_decl.name) (program.procs p))
```

```
in
  (∀ name ∈ prog_vars.
    name ∉ set (reserved_keywords @ test_keywords)) ⋀
  (∀ name ∈ proc_names.
    name ∉ set (reserved_keywords @ test_keywords)) ⋀
  (∀ fname ∈ proc_names.
    (∀ vname ∈ set (program.globals p). fname ≠ vname)))
```

Apéndice B

Reglas del CFG

```
type_synonym cfg_label = enabled × transformer

inductive cfg :: com ⇒ cfg_label ⇒ com ⇒ bool where

  Assign: cfg (x ::= a) (en_always,tr_assign x a) SKIP
| Assignl: cfg (x ::==  a) (en_always,tr_assignl x a) SKIP
| Seq1: cfg (SKIP;; c2) (en_always, tr_id) c2
| Seq2: ⟦cfg c1 a c1' ⟧ ⟹ cfg (c1;; c2) a (c1';; c2)
| IfTrue: cfg (IF b THEN c1 ELSE c2) (en_pos b, tr_eval b) c1
| IfFalse: cfg (IF b THEN c1 ELSE c2) (en_neg b, tr_eval b) c2
| While: cfg (WHILE b DO c) (en_always, tr_id)
    (IF b THEN c;; WHILE b DO c ELSE SKIP)
| Free: cfg (FREE x) (en_always, tr_free x) SKIP

| Return: cfg (Return a) (en_always, tr_return a) SKIP
| Returnv: cfg Returnv (en_always, tr_return_void) SKIP

| Callfunl: cfg (Callfunl e f params)
    (en_always, tr_callfunl proc_table e f params) SKIP
| Callfun: cfg (Callfun x f params)
    (en_always, tr_callfun proc_table x f params) SKIP
| Callfunv: cfg (Callfunv f params)
    (en_always, tr_callfunv proc_table f params) SKIP
```

Apéndice C

Demostraciones de lemas de la semántica

Demostración del lema 3.1:

Lema: (cfg_has_enabled_action).
$c \neq \texttt{SKIP} \Longrightarrow \exists\, c'\ en\ tr.\ \texttt{cfg}\ c\ (en, tr)\ c' \ \wedge\ (en\ s\ = \texttt{None} \vee en\ s\ = \texttt{Some True})$

Demostración. La prueba es por inducción en el comando. A excepción de dos casos, la prueba es demostrada automáticamente. Esos dos casos son los comandos `Seq` y `If`. En el caso de `Seq` se debe hacer una prueba por casos sobre el primer comando de la secuenciación para poder diferenciar los casos cuando es `SKIP` y cuando no. En ambos casos existe una regla en la definición de CFG que asegura que hay una arista habilitada que se puede tomar y el caso se soluciona automáticamente.

En el caso de `If` se debe hacer una prueba por casos sobre el valor de `en_pos` $b\ s$. Puede fallar y retornar un valor `None`, entonces el caso está resuelto. Si no falla entonces se debe chequear el valor retornado por la función `en_pos`. En el caso donde es `True` entonces el caso está resuelto. El caso difícil es cuando la evaluación de `en_pos` es `False`, semánticamente, esto significa que se debe tomar la rama del `else`. Por lo tanto cuando `en_pos` $b\ s$ es `False`, `en_neg` $b\ s$ siempre se evaluará como `True`, esto significa qe el comando siempre tendrá una arista habilitada que puede seguir, es decir, `en_neg` $b\ s$ y la demostración de este caso está completa. □

Demostración del lema 3.2:

Lema: (can_take_step).
¬ `is_empty_stack` $s \Longrightarrow \exists\, x.\ s\ \rightarrow\ x$

Demostración. De las suposiciones sabemos que la pila no está vacía, por lo tanto se puede reescribir el estado de la siguiente manera: $s = ((c, locals, rloc)\#\sigma, \gamma, \mu)$. Luego se puede hacer una prueba por casos. Se tiene el caso donde $c\ =$ `SKIP` y el caso donde c
neq `SKIP`.

El caso donde $c\ =$ `SKIP` es el caso en el que se está retornando de la ejecución de una función. Se tiene que la pila no está vacía, que $c\ =$ `SKIP` y que $s = ((c, locals, rloc)\#\sigma, \gamma, \mu)$. Este caso corresponde a las reglas `Return_void` y `Return_void_None` en la definición de `small_step`. En ambos casos la semántica puede tomar un paso, bien sea a un nuevo estado s' o a `None`. El caso se resuelve automáticamente por Isabelle al indicar las reglas mencionadas y las suposiciones.

El segundo caso es cuando c
neq `SKIP`. Este es el caso donde se ejecuta cualquier otro comando que no sea el retorno de una función. En este caso se utiliza el lema anterior 3.1 y se tiene que `cfg` $c\ (en, tr)\ c'$ y o la función *enabled* falla (`en` $s\ =$ `None`) o está habilidada (`en`$|\ s\ =$ `SomeTrue`).

Esto se demuestra mediante una separación por casos. En el primer caso se supone `cfg` $c\ (en, tr)\ c'$ y `en` $s\ =$ `None`. Este es el caso donde la evaluación de la función falla, se sabe que este caso toma un paso corto a `None` dado por el caso `None` de la definición de `small_step`. Entonces se ha demostrado que existe un estado tal que $s\ \rightarrow$ `None`.

En el siguiente caso se supone `cfg` $c\ (en, tr)\ c'$ y `en` $s\ =$ `SomeTrue`. Este es el caso donde hay una arista que está habilitada para ser seguida y aún se deben revisar dos casos mas. Aplicar la función *transformer* sobre el estado actualizado con el comando $(tr\ (upd\_com\ c'\ s))$ puede fallar o no. En el caso donde falla (retorna `None`) se puede tomar un paso a `None` y se habrá probado que hay un estado al que s puede tomar un paso corto, es decir `None`. En el caso donde no falla, retornará `Some` s_2. En este caso se puede tomar un paso a `Some` s_2 y se habrá probado que hay un estado al cual s puede tomar un paso corto, es decir `Some` s_2. □

Demostración del lema 3.3:

Lema: (cfg_SKIP_stuck).
¬ `cfg SKIP` $a\ c$

Demostración. La propiedad es demostrada automáticamente. □

Demostración del lema 3.4:

Lema: (ss_empty_stack_stuck).

`is_empty_stack` $s \Longrightarrow \neg\ (s\ \rightarrow\ cs')$

Demostración. La propiedad es demostrada automáticamente. □

Demostración del lema 3.5:

Lema: (ss'_SKIP_stuck).

`is_empty_stack` $s \Longrightarrow \neg\ (Somes\ \rightarrow\ cs')$

Demostración. La propiedad es demostrada automáticamente. □

Demostración del lema 3.6:

Lema: (en_neg_by_pos).

`en_neg` $e\ s =$ `map_option (HOL.Not) (en_pos` $e\ s$ `)`

Demostración. La propiedad es demostrada automáticamente desplegando las definiciones de `en_neg` y `en_pos`. □

Demostración del lema 3.7:

Lema: (cfg_determ).

`cfg` $c\ a1\ c' \wedge$ `cfg` $c\ a2\ c''$

$\Longrightarrow a1\ =\ a2 \wedge c'\ =\ c'' \vee$

$\exists\ b.\ a1\ =$ (`en_pos` b, `tr_eval` b) $\wedge\ a2\ =$ (`en_neg` b, `tr_eval` b) $\vee$

$\exists\ b.\ a1\ =$ (`en_neg` b, `tr_eval` b) $\wedge\ a2\ =$ (`en_pos` b, `tr_eval` b)

Demostración. La demostración es por inducción sobre el comando, con los casos generados automáticamente por Isabelle de las reglas de `cfg`. Todos los casos se demuestran automáticamente. □

Demostración del lema 3.8:

Lema: (lift_upd_com).

$\neg$ `is_empty_stack` $s\ \Longrightarrow$

`lift_transformer_nr` tr (`upd_com` $c\ s$) $=$

`map_option` (`upd_com` c) (`lift_transformer_nr` $tr\ s$)

Demostración. Es demostrado automáticamente al desplegar la definición de la función `lift_transformer_n`

□

Demostración del lema 3.9:

Lema: (tr_eval_upd_com).

¬ `is_empty_stack` $s \implies$

`tr_eval` e (`upd_com` c s) $=$

`map_option` (`upd_com` c) (`tr_eval` e s)

Demostración. Es demostrado automáticamente desplegando la definición de `tr_eval`. □

Demostración del lema 3.10:

Lema: (small_step_determ).

$s \rightarrow s' \wedge s \rightarrow s'' \implies s' = s''$

Demostración. La demostración es una prueba por casos sobre la semántica de pasos cortos. Se obtienen 4 casos, cada uno correspondiente a cada regla en la semántica de pasos cortos. Los objetivos de prueba generados por las reglas `Return_void` y `Return_void_None` se resuelven automáticamente. Los objetivos de prueba generados por las reglas `Base` y `None` se resuelven automáticamente luego de agregar los lemas 3.6 y 3.9. □

Demostración del lema 3.11:

Lema: (small_step'_determ).

$s \rightarrow' s' \wedge s \rightarrow' s'' \implies s' = s''$

Demostración. La demostración es una prueba por casos sobre la semántica de pasos cortos. Se demuestra automáticamente mediante el uso del lema 3.10. □

Apéndice D

Demostraciones de lemas para el interpretador

Demostración del lema 3.12

Lema: (fstep1).

$s \rightarrow s' \Longrightarrow \texttt{fstep}\ s = s'$

Demostración. La demostración es por inducción sobre la semántica de pasos cortos. □

Luego se considera la dirección opuesta:

Demostración del lema 3.13

Lema: (fstep2).

$\neg\ \texttt{is\_empty\_stack}\ s \Longrightarrow s \rightarrow (\texttt{fstep}\ s)$

Demostración. La demostración se hace automáticamente mediante una prueba por casos sobre el resultado de "`tr_return_void` s" y utilizando los lemas 3.2 y 3.12. □

Demostración del lema 3.16

Lema: (interp_unfold).

`interp proc_table` cs = (

`if is_term` (`Some` cs) `then Some` cs

`else do{` $cs \leftarrow$ `fstep proc_table` cs; `interp proc_table` cs `})`

Demostración. La demostración es hecha automáticamente. □

Demostración del lema 3.17

Lema: (None_star_preserved).

$\texttt{None} \rightarrow * z \longleftrightarrow z = \texttt{None}$

Demostración. La demostración es por inducción sobre la clausura reflexivo transitiva (`star`). Los objetivos se resuelven automáticamente. □

Demostración del lema 3.18

Teorema: (interp_correct).

$\texttt{terminates}\ cs \implies (\texttt{yields}\ cs\ cs') \longleftrightarrow (cs' = \texttt{interp}\ \texttt{proc\_table}\ cs)$

Demostración. La demostración se hace suponiendo el antecedente y demostrando cada dirección de la igualdad por separado. □

Apéndice E

Pretty Printing

E.1. Palabras

La primera instanciación de `shows` que se debe hacer es aquella para las palabras de modo que sea posible imprimir valores de este tipo. Para hacer *pretty printing* de un valor del tipo palabra simplemente se hace *casting* de ese valor a un tipo entero predefinido por Isabelle/HOL y se utiliza la función `show` para ese tipo. Como resultado, las palabras serán impresas como enteros con signo.

E.2. Valores

Aunque el tipo de datos `val` y las valuaciones no son utilizadas en el proceso de generación de código, se considera útil tener un mecanismo para hacer *pretty printing* de los mismos con fines de depuración.

E.2.1. Tipo val

El tipo `val` es el primer tipo de datos definido por el usuario para el que se presenta una instanciación. En la tabla E.1 se encuentra la equivalencia entre la sintaxis abstracta para el tipo `val` y la representación en cadena de caracteres. Nótese que w, $base$ y ofs representan palabras, naturales y enteros, por lo tanto sus funciones `show` serán utilizadas para obtener su representación en cadena de caracteres, por ejemplo, la representación en cadena de caracteres de `I` 42 sería "42", la representación en cadena de caracteres de `A` (4, 2) sería "4[2]". Se puede realizar `show` de una lista de valores al hacer `show` de cada valor en una notación de listas, es decir, $[vname = valor, \ldots, vname_n = valor_n]$.

Se rodean con cuñas los parámetros (" < > ") para indicar que lo que está dentro de ellos es una representación en forma de cadena de caracteres resultante de aplicar una función `show` en el parámetro y se continuará usando esta notación a lo largo del documento.

Sintaxis abstracta	**Representación en cadena de caracteres**
`NullVal`	null
`I` w	$< w >$
`A` $(base,\ ofs)$	$< base >[< ofs >]$

Tabla E.1: Traducción del tipo val

E.2.2. Tipo val option

Un `val option` tiene el significado semántico de un valor inicializado y no inicializado. La tabla E.2 muestra la equivalencia entre la sintaxis abstracta y la representación en cadena de caracteres. Un valor inicializado simplemente será la cadena de caracteres resultante de llamar a la función `show` sobre ese valor, mientras que la representación de un valor no inicializado será un $''?''$ para representar que el valor es aun desconocido.

Sintaxis abstracta	**Representación en cadena de caracteres**
`Some` v	$< v >$
`None`	?

Tabla E.2: Traducción del tipo val option

E.2.3. Valuaciones

Para representar una valuación se necesita un parámetro extra: una lista de nombres de variables, esta lista tendrá los nombres de las variables para los cuales se quiere imprimir su valor en la valuación. La valuación será impresa con el siguiente formato:

$$[< vname_0 >=< valor_0 >, < vname_1 >=< valor_1 >, \ldots, < vname_n >=< valor_n >]$$

Por ejemplo, si se toma la valuación $[foo \mapsto \texttt{Some}(\texttt{I}\ 15), bar \mapsto None, baz \mapsto \texttt{Some}(\texttt{A}(1, 9))]$ y la lista de variables $[foo, bar, baz]$ entonces se obtiene la siguiente representación para la valuación: $[foo\ =\ 15,\ bar\ =\ ?,\ baz\ =\ 1[9]]$.

E.3. Memoria

La memoria, al igual que los valores, no es un componente necesario para generar programas en C. No obstante, tener una manera de hacer *pretty printing* de la memoria en cierto estado es útil al de realizar depuración. Para imprimir la memoria se debe saber como hacer `show` del contenido de un bloque y de todo el bloque.

Cuando se accede a un bloque a memoria se puede obtener tanto el contenido del bloque o un valor `None` que indica un bloque libre en memoria. La tabla E.3 muestra la representación en cadena de caracteres de esto. Nótese que en el caso de un valor `None` se imprime la cadena `free` rodeada de cuñas, siendo las mismas parte de la representación y una excepción a la notación descrita anteriormente. Si el contenido del bloque es una lista de valores entonces se imprime la lista de valores.

Sintaxis abstracta	**Representación en cadena de caracteres**
`Some` content	$< content >$
`None`	<free>

Tabla E.3: Traducción del contenido de un bloque

Un bloque completo se imprime de la siguiente manera:

$$< base > : < contenido\_bloque >$$

donde *base* es el primer componente de una dirección con la que se accede al bloque y *contenido_bloque* es la representación en cadena de caracteres del contenido en el bloque número *base*.

Para imprimir la memoria completa se hace `show` de cada bloque existente en memoria. Por ejemplo, la representación en cadena de caracteres del estado de memoria: [`Some` [`I` 13, `None`], $None$, `Some` [`A`(2 sería:

$$0 : [13, ?]$$
$$1 : < free >$$
$$2 : [2[3], ?, 56]$$

E.4. Expresiones

Operaciones unarias y binarias Al hacer *pretty printing* de operaciones binarias se utilizan paréntesis alrededor de cada expresión que se imprime. Esto, naturalmente, generará más paréntesis de los necesarios pero se toma esta decisión para asegurar que el orden de evaluación se mantenga igual al previsto y que no se obtengan diferentes órdenes de evaluación debido a la precedencia de los operadores.

Un operador binario se imprime usando notación de infijo: ($< operando_1 >< operador >< operando_2 >$). Ejemplos de esto se muestran en la tabla E.4.

Sintaxis abstracta	**Representación en cadena de caracteres**
`Plus (Const 11) (Const 11)`	(11 + 11)
`Subst (Const 9) (Const 5)`	(9 − 5)
`Mult (Const 2) (Const 3)`	(2 ∗ 3)

Tabla E.4: Ejemplos de *pretty printing* de operadores binarios

Un operador unario se imprime utilizando notación de prefijo: $< operador >$ ($< operando >$). Nótese que se rodea entre paréntesis el operando con el fin de garantizar la correcta precedencia en las operaciones. Ejemplos de esto se muestran en la tabla E.5.

Sintaxis abstracta	**Representación en cadena de caracteres**
`Minus (Const 11)`	− (11)
`Not (Const 0)`	! (0)

Tabla E.5: Ejemplos de *pretty printing* de operadores unarios

Expresiones En la tabla E.6 se presenta la representación en cadena de caracteres para cada una de las expresiones. Se utilizan operaciones simples como operandos tales como variables o valores constantes por simplicidad, sin embargo las expresiones pueden estar compuestas por términos más complejos.

E.5. Comandos

Primeramente, se necesita un mecanismo que permita imprimir comandos indentados para facilitar la generación de código. Se definen dos abreviaciones auxiliares que imprimen espacios en blanco para indentación al inicio de un término. La razón por la que se definen dos

Sintaxis abstracta	**Representación en cadena de caracteres**
`Const` 42	42
`Null`	(`intptr_t` *) 0
`V` x	x
`Plus` (`Const` 2) (`Const`5)	(2 + 5)
`Subst` (`Const` 9 (`Const`5)	(9 − 5)
`Minus` (`Const` 9)	(−9)
`Div` (`Const` 8) (`Const` 4)	(8 / 4)
`Mod` (`Const` 8) (`Const` 4)	(8 % 4)
`Mult` (`Const` 9) (`Const` 3)	(9 ∗ 3)
`Less` (`Const` 7) (`Const` 9)	(7 < 9)
`Not` (`Const` 0)	! (0)
`And` (`Const` 1) (`Const` 1)	(1 && 1)
`Or` (`Const` 1) (`Const` 0)	(1 ||0)
`Eq` (`Const` 6) (`Const` 4)	(6 == 4)
`New` (`Const` 9)	(`intptr_t`) __`MALLOC`(`sizeof`(`intptr_t`) ∗ (9))
`Deref` (`V` foo)	∗((`intptr_t` ∗) foo)
`Ref` (`V` foo)	((`intptr_t` ∗) &(foo))
`Index` (`V` bar) (`Const` 3)	((`intptr_t` ∗) bar[3])
`Derefl` (`V` foo)	(∗((`intptr_t` ∗) foo))
`Indexl` (`V` bar) (`Const` 3)	((`intptr_t` ∗) bar[3])

Tabla E.6: Ejemplos de *pretty printing* para expresiones

abreviaciones es porque una de ellas también imprime un terminador ";" luego del término, mientras que la otra no.

Las llamadas a funciones se imprimen siguiendo el siguiente formato: $< nombre\_funcion >$ ([$< argumento_0, argumento_1, \ldots, < argumento_n >$]) donde los corchetes ([]) indican que los argumentos son opcionales.

Los comandos en Chlose se imprimen como se muestra en los ejemplos de la tabla E.7. Se utiliza $''$ para indicar una cadena vacía. El nivel correcto de indentación se indica en los parámetros de la función que se encarga de realizar el *pretty printing*, acá esos son omitidos y en su lugar se indica donde la indentación aumenta. La indentación aumenta al imprimir comandos que se encuentren dentro de un bloque, por ejemplo: las ramas de un condicional o el cuerpo de un ciclo.

Sintaxis abstracta	**Representación en cadena de caracteres**
SKIP	″
Derefl foo ::== Const 4	$*$((intptr_t $*$) foo) = 4;
foo ::= Const 4	foo = 4;
c_1;; c_2	$< c_1 >< c_2 >$

IF (V b) THEN
foo ::= Const 4
ELSE
SKIP
	if (b) {
	foo = 4;
	}

If (V b) THEN
foo ::= Const 4)
ELSE
bar ::= Const 3)
	if (b) {
	foo = 4;
	} $else$ {
	bar = 3;
	}

While (V b) DO
foo ::= Const 4
	while (b) {
	foo = 4;
	}

FREE (Derefl foo)	free (& ($*$((intptr_t $*$) foo));
RETURN (V foo)	return (foo));
RETURNV	return;
(Derefl foo) ::== bar ([V baz, Const 4])	$*$((intptr_t $*$) foo) = bar(baz, 4);
foo ::= bar ([Const 65])	foo = bar(65);
CALL bar ([])	bar();

Tabla E.7: Ejemplos de *pretty printing* para comandos

E.6. Declaraciones de funciones

A continuación se define como se imprimen las declaraciones de funciones. Para hacerlo, se imprime la definición de una función de acuerdo al siguiente formato:

intptr_t $< nombre\_de\_funcion >$ (intptr_t $< nombre\_arg_0 >$, ...,
intptr_t $< nombre\_arg_n >$) {
intptr_t $< var\_local_0 >$
⋮
intptr_t $< var\_local_n >$
$< cuerpo >$
}

El retorno, los argumentos y las variables locales es de tipo `intptr_t` dado que, como se mencionó anteriormente, solo se tiene un tipo en el proceso de traducción y se realizan *casts*

hacia y desde apuntadores cuando son necesarios.

Un ejemplo de la traducción de la declaración de una función para una función factorial se encuentra en la tabla E.8. En este ejemplo se evita el uso de ″ ″ para representar cadenas de caracteres en Isabelle/HOL y simplemente se escribe la cadena de caracteres sin las comillas.

Sintaxis abstracta	**Representación en cadena de caracteres**

`definition factorial_decl :: fun_decl`
`where "factorial_decl ≡`
`  ( fun_decl.name = fact,`
`  fun_decl.params = [n],`
`  fun_decl.locals = [r, i],`
`  fun_decl.body =`
`  r ::= (Const 1);;`
`  i ::= (Const 1);;`
`  (WHILE`
`  (Less (V i) (Plus (V n) (Const 1)))`
`  DO`
`  (r ::= (Mult (V r) (V i));;`
`  i ::= (Plus (V i) (Const 1)))`
`  );;`
`  RETURN (V r)`
`  )"`
	`intptr_t fact(intptr_t n) {`
	`  intptr_t r;`
	`  intptr_t i;`
	`  r = (1);`
	`  i = (1);`
	`  while ((i) < ((n) + (1))) {`
	`  r = ((r) * (i));`
	`  i = ((i) + (1));`
	`  }`
	`  return(r);`
	`}`

Tabla E.8: Ejemplo de *pretty printing* de la declaración de una función factorial

E.7. Estados

Al ejecutar un programa dentro del ambiente de Isabelle/HOL a menudo se quieren inspeccionar los estados. En esta sección se define una forma sencilla de inspeccionar los estados al tener una representación en cadena de caracteres para los mismos.

Primero se describe como se imprime una ubicación de retorno. Se instancia la clase `show` para el tipo `return_loc`. En la tabla E.9 se encuentra la equivalencia entre la sintaxis abstracta para el tipo `return_loc` y su representación en cadena de caracteres. Donde $< base >$ y $< ofs >$ son el resultado de aplicar la función `show` sobre $base$ y ofs y $< invalid >$ es una cadena de caracteres literal que incluye las cuñas.

Luego se describe como se imprime la pila. Un solo marco de pila se imprime al imprimir el comando, la lista de variables locales y la ubicación de retorno esperada con el siguiente formato: $rloc = < rloc >$ separados por un salto de línea. Para imprimir la pila completa, se imprime cada marco de pila separado por "----------".

Sintaxis abstracta	**Representación en cadena de caracteres**
`Ar` $(base,\ ofs)$	$< base >[< ofs >]$
`Vr` w	w
`Invalid`	<invalid>

Tabla E.9: Traducción del tipo de una ubicación de retorno

Para imprimir un estado se debe dar a la función `show` una lista de nombres de variables, la cual será utilizada para imprimir la valuación para las variables globales y para las locales en cada marco de pila. Un estado se imprime al imprimir la pila, los valores de las variables globales y, finalmente, la memoria, separados por "`==========`" Un ejemplo de *pretty printing* para un estado se muestra en la tabla E.10.

Sintaxis abstracta	**Representación en cadena de caracteres**

([(x ::= `Const` 4;; y ::= `Const` 25,
[z ↦ `Some` (`I` 0)], `Invalid`),
[(x ::= `Const` 3;; y ::= `Const` 43,
[z ↦ `Some` (`I` 6)], `Vr`foo)],
[x ↦ `Some` (`I` 3), y ↦ `Some` (`I` 8),
foo ↦ `Some` (`I` 0)],
[`None`, `Some` [`Some` (`I` 44), `Some` (`A` (2, 0))],
`Some` [`Some` (`I` 78)]
	`x = (4);`
	`y = (25);`
	`---------------`
	`[z = 0]`
	`---------------`
	`rloc = <invalid>`
	`x = (5);`
	`y = (43);`
	`---------------`
	`[z = 6]`
	`---------------`
	`rloc = foo`
	`==========`
	`[x = 3, y = 8, foo = 0]`
	`==========`
	`1: <free>`
	`2: [44, *2[0]]`
	`3: [78]`

Tabla E.10: Ejemplo de *pretty printing* para un estado

Apéndice F

Generación de código en C

```
fun export_c_code (SOME code) rel_path name thy =
  let
    val str = code |> String.implode;
  in
    if rel_path="" orelse name="" then
      (writeln str; thy)
    else let
      val base_path = Resources.master_directory thy
      val rel_path = Path.explode rel_path
      val name_path = Path.basic name |> Path.ext "c"

      val abs_path = Path.appends [base_path, rel_path, name_path]
      val abs_path = Path.implode abs_path

      val _ = writeln ("Writing to file " ^ abs_path)

      val os = TextIO.openOut abs_path;
      val _ = TextIO.output (os, str);
      val _ = TextIO.flushOut os;
      val _ = TextIO.closeOut os;
      in thy end
    end
| export_c_code NONE _ _ thy =
    (error "Invalid program, no code is generated."; thy)
```

Apéndice G

DFS para generación de pruebas

```
context fixes μ :: mem begin

partial_function (option) dfs
  :: "nat set ⇒ addr ⇒ string ⇒ (nat set × test_instr list) option"
  where
  [code]: "dfs D a ca = do {
    let (base,ofs) = a;

    case μ!base of
      None ⇒ Some (D,[])
    | Some b ⇒ do {
        let ca = adjust_addr ofs ca;
        if base ∉ D then do {
          let D = insert base D;
          let emit = [Discover ca base];

          fold_option (λi (D,emit). do {
            let i=int i;
            let cval = (ofs_addr i (base_var_name base));
            case b!!i of
              None ⇒ Some (D,emit)
            | Some (I v) ⇒ Some (D,emit @ [Assert_Eq cval v])
            | Some (NullVal) ⇒ Some (D,emit @ [Assert_Eq_Null cval] )
            | Some (A addr) ⇒ do {
                (D,emit') ← dfs D addr cval;
                Some (D,emit@emit')
              }
          })
            [0..<length b]
            (D,emit)

        } else do {
```

```
          Some (D,[Assert_Eq_Ptr ca base])
        }
      }
  }"
end
```

Apéndice H

Archivo de cabecera test_harness.h

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <inttypes.h>
#include "hashset.h"

hashset_t __test_harness_discovered;
int __test_harness_num_tests = 0;
int __test_harness_passed = 0;
int __test_harness_failed = 0;

#define __TEST_HARNESS_DISCOVER(addr, var)
  hashset_add(__test_harness_discovered, addr); var = addr;

#define __TEST_HARNESS_ASSERT_EQ(var, val)
  ++__test_harness_num_tests;
  (var != val) ? ++__test_harness_failed : ++__test_harness_passed;

#define __TEST_HARNESS_ASSERT_EQ_NULL(var)
  ++__test_harness_num_tests;
  (var != NULL) ? ++__test_harness_failed : ++__test_harness_passed;

#define __TEST_HARNESS_ASSERT_EQ_PTR(var, val)
  ++__test_harness_num_tests;
  (var != val) ? ++__test_harness_failed : ++__test_harness_passed;
```

Apéndice I

Shell script para la ejecución de pruebas

```
#!/bin/bash

TEST_NAMES=(bubblesort_test count_test fact_test fib_test
  mergesort_test min_test occurs_test quicksort_test selection_test
  strlen_test plus_test subst_test outer_scope_test local_scope_test
  global_scope_test global_scope2_test mod_test div_test mult_test
  less_test and_test or_test not_test eq_test new_test deref_test
  while_test returnv_test linked_list_test cyclic_linked_list_test)

for test_name in ${TEST_NAMES[@]}
do
  res=$(./${test_name});
  ret=$?

if [[ ${res} == Failed* ]];
  then
    echo -e "\e[31mFAILED: \e[39m${res}"
else
  case ${ret} in

  1) echo -e "\e[31mError\e[39m (general error)
    occurred in the execution of ${test_name}";;
  2) echo -e "\e[31mError\e[39m (misuse of shell builtins)
    occurred in the execution of ${test_name}";;
  3) echo -e "\e[31mMemory allocation error\e[39m ocurred
    in execution of ${test_name}";;
  126) echo -e "\e[31mError\e[39m (command invoked cannot execute)
    occurred in the execution of ${test_name}";;
  127) echo -e "\e[31mError\e[39m (command not found)
    occurred in the execution of ${test_name}";;
  128) echo -e "\e[31mError\e[39m (invalid argument given to exit)
    occurred in the execution of ${test_name}";;
  130) echo -e "\e[31mError\e[39m (program terminated by Ctrl+C)
    occurred in the execution of ${test_name}";;
  139) echo -e "\e[31mSegmentation fault\e[39m ocurred
    in execution of ${test_name}";;
  *) echo ${res};;
  esac
fi
done
```

Apéndice J

Ejemplo de *pretty printing*: Factorial

Factorial se define de la siguiente manera en Isabelle:

```
definition factorial_decl :: fun_decl
  where "factorial_decl ≡
    ( fun_decl.name = fact,
      fun_decl.params = [n],
      fun_decl.locals = [r, i],
      fun_decl.body =
        r ::= (Const 1);;
        i ::= (Const 1);;
        (WHILE (Less (V i) (Plus (V n) (Const 1))) DO
          (r ::= (Mult (V r) (V i));;
          i ::= (Plus (V i) (Const 1)))
        );;
        RETURN (V r)
    )"

definition main_decl :: fun_decl
  where "main_decl ≡
    ( fun_decl.name = main,
      fun_decl.params = [],
      fun_decl.locals = [],
      fun_decl.body =
        n ::= Const 5;;
        r ::= fact ([V n])
    )"

definition p :: program
  where "p ≡
    ( program.name = fact,
```

```
    program.globals = [n, r],
    program.procs = [factorial_decl, main_decl]
  )"
```

Mediante el uso del proceso de *pretty printing* se genera el siguiente código en lenguaje C:

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <stdint.h>
#include "../test_harness.h"
#include "../malloc_lib.h"
#ifndef INTPTR_MIN
  #error ("Macro INTPTR_MIN undefined")
#endif
#ifndef INTPTR_MAX
  #error ("Macro INTPTR_MAX undefined")
#endif
#if ( INTPTR_MIN + 1 != -9223372036854775807 )
  #error ("Assertion INTPTR_MIN + 1 == -9223372036854775807 failed")
#endif
#if ( INTPTR_MAX != 9223372036854775807 )
  #error ("Assertion INTPTR_MAX == 9223372036854775807 failed")
#endif


intptr_t n;
intptr_t r;

intptr_t fact(intptr_t n) {
  intptr_t r;
  intptr_t i;
  r = (1);
  i = (1);
  while ((i) < ((n) + (1))) {
    r = ((r) * (i));
    i = ((i) + (1));
  }
  return(r);
}

intptr_t main() {
  n = (5);
  (r) = (fact(n));
}
```

Nótese que la verificación de los límites para los enteros para el límite inferior tiene una solución temporal ya que el valor de `INTPTR_MIN` excede el límite superior para los enteros

del preprocesador lo cual causa *overflow* y una advertencia al compilar. [1] Para eliminar esta advertencia, se compara el valor de `INTPTR_MIN + 1` al de `INT_MIN + 1`.

[1] Interpreta el número negativo como $-(numero)$ y produce una advertencia que indica que no puede representar *numero*.

Apéndice K

Generación de código en C con pruebas

```
fun generate_c_test_code (SOME (code,test_code)) rel_path name thy =
  let
    val code = code |> String.implode
    val test_code = test_code |> String.implode
  in
    if rel_path="" orelse name="" then
      (writeln (code ^ " <rem last line> " ^ test_code); thy)
    else let
      val base_path = Resources.master_directory thy
      val rel_path = Path.explode rel_path
      val name_path = Path.basic name |> Path.ext "c"

      val abs_path = Path.appends [base_path, rel_path, name_path]
      val abs_path = Path.implode abs_path

      val _ = writeln ("Writing to file " ^ abs_path)

      val os = TextIO.openOut abs_path;
      val _ = TextIO.output (os, code);
      val _ = TextIO.flushOut os;
      val _ = TextIO.closeOut os;

      val _ = Isabelle_System.bash ("sed -i '$d ' " ^ abs_path);

      val os = TextIO.openAppend abs_path;
      val _ = TextIO.output (os, test_code);
      val _ = TextIO.flushOut os;
      val _ = TextIO.closeOut os;
    in thy end
```

```
    end

  | generate_c_test_code NONE _ _ _ =
      error "Invalid program or failed execution"

fun expect_failed_test (SOME _) = error "Expected Failed test"
  | expect_failed_test NONE = ()
```