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, @mes de 2015

UNIVERSIDAD SIMÓN BOLÍVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERÍA DE LA COMPUTACIÓN

ACTA FINAL PROYECTO DE GRADO

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

Presentado por:
Gabriela Limonta

Este Proyecto de Grado ha sido aprobado por el siguiente jurado examinador:

Federico Flaviani

@jurado1

@jurado2

Sartenejas, @día de @mes de 2015

Resumen

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Palabras clave: @palabra1, @palabra2, @palabra3.

Agradecimientos

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Índice general

Índice de figuras

Índice de Tablas

Acrónimos y Símbolos

SIGLAS	**S**iglas **I**sla **G**rafo **L**aos **A**ve **S**erpiente
ACM	**A**ssociation for **C**omputing **M**achinery

$\iff$	doble implicación, si y sólo si
$\Rightarrow$	implicación lógica
$[u := v]$	sustitución textual de u por v

Dedicatoria

A @personasImportantes, por @razonesDedicatoria.

Capítulo 1

Introduction

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 [ISO07] es propensa a ambigüedades debido al uso del idioma inglés para describir el comportamiento de un programa. El uso de "formal mathematical constructs" eliminaría estas ambigüedades, aunque la formalización de la semántica para la totalidad del lenguaje C no es una tarea facil 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 algorítmos tales como algorítmos 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 “deterministic”, 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 “test harness” y un “test suite” 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 “testing” 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 “test harness” 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 programción

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 [Ten91], 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 [NN07], el carácter riguroso de este estudio se debe al hecho de que puede revelar ambigüedades o complejidades subyacientes 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 [NN07]. Se toma el siguiente programa

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

donde ":=" es una asignación a una variable y ";" es "sequential composition". 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 [NN07]. Dado un programa, su explicacion 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 [NK14].

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:

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

donde se utilizan las siguientes abreviaciones:

$$\begin{aligned} s_0 &= [x \mapsto 5, y \mapsto 7, z \mapsto 0] \\ s_3 &= [x \mapsto 7, y \mapsto 5, z \mapsto 5] \end{aligned}$$

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

$$\cfrac{\cfrac{\langle z := x, s_0\rangle \rightarrow s_1 \qquad \langle x := y, s_1\rangle \rightarrow s_2}{\langle z := x; x := y, s_0\rangle \rightarrow s_2} \qquad \langle y := z, s_2\rangle \rightarrow s_3}{\langle z := x; x := y; y := z, s_0\rangle \rightarrow s_3}$$

donde se utilizan las siguientes abreviaciones:

$$\begin{aligned} s_0 &= [x \mapsto 5, y \mapsto 7, z \mapsto 0] \\ s_1 &= [x \mapsto 5, y \mapsto 7, z \mapsto 5] \\ s_2 &= [x \mapsto 7, y \mapsto 7, z \mapsto 5] \\ s_3 &= [x \mapsto 7, y \mapsto 5, z \mapsto 5] \end{aligned}$$

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. Finalemnte, 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 explicitamente inspeccionar ejecuciones parciales [NK14]

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":

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

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 $y := z$, despues 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 enfoce hacia el *efecto* de ejecutar el programa. Este enfoque sirve de ayuda pues da un *significado* a los programas en un lenguaje [NK14]. 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 facilmente adaptado para representar algunas propiedades de los programas. Ejemplos de esto son inicialización de variables, “constant” folding y alcanzabilidad [NN07].

Semántica Axiomática

Tambien 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 [NK14].

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

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

donde $\{x = n \land y = m\}$ y $\{x = m \land y = n\}$ 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 [NK14]. El siguiente “arbol de demostración” puede expresar una prueba de la propiedad de correctitud parcial anterior.

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

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 automátizadas 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 [NPW15], se puede estar seguro de que nos resultados demostrados son correctos. En el entorno de Isabelle/HOL se pueden hacer definiciones lógicas y demostrar lemmas 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* [NK14] en el que se muestra una introducción a Isabelle/HOL y demotración de teoremas.

Por otra parte, Isabelle/HOL también cuenta con herramientas de generación de código [Haf15] 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. Tambien, Isabelle/HOL permite que código escrito en Isabelle/ML esté embebido en las teorías [Wen15], 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 fururo 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 Chloa 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, asi 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.

Primero, se hablará de la formalización de C en HOL de Michael Norrish [Nor98]. Este trabajo está relacionado al presente porque formaliza la semántica de un subconjunto de C utilizando HOL y el subconjunto de C para el cual se formaliza la semántica es mas grande que el presentado en este trabajo.

Luego, se discutirá el modelo de memoria utilizado en la verificación formal del compilador CompCert [LB08] En el presente trabajo se adopta el modelo de memoria utilizado por el compilador verificado de C del proyecto CompCert.

Finalmente, se mencionará el proyecto Autocorres [GLAK14], el cual tiene como fin el abstraer la semántica de bajo nivel de C a una representación de más alto nivel. Este proyecto traduce código de lenguaje C a la lógica de un probador de teoremas con el fin de poder probar propiedades del código fuente en C.

2.1. Formalización de C en HOL

El trabajo de Michael Norrish [Nor98] 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 [NS14].

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.

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. CompCert Compiler's Memory Model

El compilador CompCert es un compilador optimizador formalmente verificado que traduce código del subconjunto Clight del lenguaje de programación C [BL09] a código ensamblador para PowerPC. Una descripción del compilador CompCert se encuentra disponible en el capítulo 4 de [BJLM13]. CompCert compila código fuenta de la semántica de Clight a código ensamblador preservando la semántica del lenguaje original. Para ralizar esta traducción se necesitan varios lenguajes,

[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 funcion.

así como un modelo de memoria que permita el rrazonamiento 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” o “partial overlapping” 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 [LB08] 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 offsets 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 [dt15]. 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 accesar 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 detras 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 “top-down” donde se tiene la intención de generar códigio en C de una especificación formal. Existe otra dirección que es relevante mencionar. El proyecto AutoCorres [GLAK14] “parses” 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 “abstraction” del “heap” que permite el razonamiento sobre memoria para “type-safe functions” así como una “abstraction” para palabras que permite que palabras de la máquina puedan ser “abstracted” a números naturales y enteros, de modo que sea posible razonar acerca de los mismos a este nivel.

Es relevante considerar este enfoque “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 “high-assurance systems”.

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. Cada una de estas instrucciones representa una instrucción que la máquina ejecutará. 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 expressiones 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;
```

```
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: Chloe expressions

```
*baz = 1;
```

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 y expresioness del lado derecho. En las siguientes secciones al referirse a expresiones del lado derecho se utilizará simplemente el nombre expresiones y se utilizará expresion LHS (por sus siglas en ingles *left-hand-side*) en lugar de 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 "dereferencing", "referencing" y acceso a arreglos, estos son, en C, los operadores `*`, `&` y `[]`, respectivamente. Finalmente, como expresiones LHS se

```
  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: Integer lower and upper bounds

tienen “dereferencing” y accesso a arreglos.

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 parametro 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 parametro puede ser cambiado con el fin de cumplir con los requerimientos de la arquitectura.

Tambien, 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* [Loc13].

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 `(block_id, offset)`. 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 [LB08], 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 :: 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: Memory Management operations

- `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 oigual 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.

- `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 "non-deterministically" que 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.

Valuations We define the type for a valuation as follows:

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

A valuation is a function that maps a variable name to a value. The return type is `val option option` to model the following states of a variable's value: undefined, uninitialized and holding a value. Therefore, given a variable name, this function can yield one of the following three results:

- `None`, which represents a variable that is undefined.

- `Some None`, which represents a variable that is defined but uninitialized.
- `Some` v, which represents a defined initialized variable that holds the value v.

Visible States When we execute a command, this command can only *see* certain part of the state. The part of the state a command can see is the one consisting of the variables that are local to the function that is being executed, the global variables and the memory. This part of the state is precisely what we call a *visible state*. We can define a visible state as the part of the state a single command can view and modify. The defined type for it is as follows:

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

A visible state is a tuple that contains a valuation function for the local variables, a valuation function for the global variables and the dynamic memory of the program.

Now we can proceed to introduce the semantics for the expressions in Chloe. As we said before the semantics of an expression is its value and its effect on a program state, therefore we have defined two evaluation functions, one that computes the value of an expression and one that computes the value of a LHS expression. These functions are defined as follows:

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

where `eval`, given an expression and a visible state, will yield the value of that expression and the visible state resulting after evaluation of that expression. `eval_l`, given a LHS expression and a visible state, will yield the value of that expression (which must be an address) and the state resulting after the evaluation of the LHS expression. Notice that the resulting type for these evaluation functions is an `option` type. This is because these functions can fail, a failure can happen anywhere in the evaluation of the expression and if a failure is found then it will propagate until the whole expression evaluation returns a `None` value indicating an error in evaluation.

Expression evaluation might fail for several reasons which include, but are not limited to, variable undefinedness, an operation failing because it has some illegal operand, trying to access an invalid part of the memory and overflow. Therefore

```
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: Auxiliary functions for eval and eval_l

if there is an error early in the expression evaluation semantics, it will be detected and propagated as a `None` value which indicates an error state.

The `eval` and `eval_l` functions depend on several other defined functions in order to properly compute the values of expressions. A definition of all the auxiliary functions for `eval` and `eval_l` is given in figure 3.4. Except for `div_towards_zero` and `mod_towards_zero`, each of these operations can fail, therefore their return type is τ `option`. The values of that type are either `None` when the operation fails or `Some`(v) where v is of type τ.

The functionalities of the auxiliary functions are the following:

- The function `detect_overflow` detects integer overflow. It takes an Isabelle predefined integer value as a parameter and checks for overflow with the bounds described in figure 3.2. This function will fail whenever overflow is detected. Upon successful execution the function will yield the value corresponding to the given integer parameter.

- The function `read_var` computes the value of a variable. This function will fail whenever the variable name given as a parameter corresponds to an undefined variable. Upon successful execution the function will yield the value of the variable. In order to compute the value of said variable this function checks the local valuation in the visible state and proceeds to yield the value of the variable if it is defined there. In the case where the variable is not defined in the local scope, the function will proceed to check the global scope and yield the value of the variable.

- The function `plus_val` computes the value of an addition between two values. This function will fail whenever overflow is detected or when anything different than two integers or an address and an integer (in that specific order) are given as parameters to the function. Upon successful execution with two integer values the function will yield an integer value corresponding to the addition of those operands. Upon successful execution with an address and an integer the function will yield an address value corresponding to adding the integer offset to the original address value.

- The function `subst_val` computes the value of a subtraction between two values. This function will fail whenever overflow is detected or when anything different than two integers or an address and an integer (in that specific order) are given as parameters to the function. Upon successful execution with two integer values the function will yield an integer value corresponding to the subtraction of those operands. Upon successful execution with an address and an integer the function will yield an address value corresponding to subtracting the integer offset to the original address value.

- The function `minus_val` computes the value of the unary minus operation over a value. This function will fail whenever overflow is detected or a value different from an integer is given as a parameter. Upon successful execution the function will yield an integer value corresponding to the result of negating the value given as a parameter.

- The function `div_towards_zero` performs integer division with truncation towards zero.

- The function `div_val` computes the value of division between two values. This function will fail whenever either overflow or division by zero are detected or when the parameters given to the function are different from integers. Upon successful execution the function will yield an integer value corresponding to the result of performing integer division on the function operands.

- The function `mod_towards_zero` performs the modulo operation with truncation towards zero.

- The function `mod_val` computes the value of performing the modulo operation between two values. This function will fail whenever either overflow or modulo by zero are detected or when the parameters given to the function are different from integers. Upon successful execution the function will yield an integer value corresponding to the result of performing integer modulo on the function operands.

- The function `mult_val` computes the value of a multiplication between two values. This function will fail whenever overflow is detected or when anything different from integers are given as parameters to the function. Upon successful execution the function will yield an integer value corresponding to the multiplication of the function operands.

- The function `less_val` computes the value of performing the less than operation between two values. This function will fail whenever anything different from integers are given as parameters to the function. Upon successful execution the function will yield an integer value of `I` 1 when the first operand is smaller than the second one, and an integer value of `I` 0 otherwise.

- The function `not_val` computes the value of performing logical negation over a value. This function will fail whenever a parameter different from an integer is given. Upon successful execution the function will yield an integer value of `I` 1 when the operand is an integer of value `I` 0, and an integer value of `I` 0 when the given operand is any integer value different from `I` 0.

- The function `to_bool` yields an Isabelle predefined boolean given a value. This function is used to compute short-circuit evaluation on the `And` and `Or` operations. This function will fail whenever the parameter is anything different from an integer. Upon successful execution the function will yield `False` when the given parameter has a value equal to `I` 0, it will yield `True` otherwise.

- The function `eq_val` computes the value of equality comparison between to values. This function will fail whenever anything different than two integers or two addresses are given as parameters to the function. Upon successful execution with two integer values the function will yield an integer value of `I` 1 if both operands were equal and an integer value of `I` 0 otherwise. Upon successful execution with two address values the function will yield an integer value of `I` 1 if both addresses were equal and an integer value of `I` 0 otherwise. Two addresses are regarded to as equal whenever both components of the address tuple are equal.

```
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: Chloe commands

- The functions `new_block` and `load` are the ones explained earlier in section 3.1.1 that allocate a new block of memory and load a value from memory, respectively.

3.2. Commands

In this next section we will discuss in detail the syntax and semantics of Chloe's commands as well as functions and programs written in the language. Additionally, we will discuss some restrictions the semantics assumes.

3.2.1. Syntax

Chloe contains the following constructs: assignments, sequential composition, conditionals, while loops, SKIP,[1] deallocation of memory, return statements and functions. The expressions are the ones described in the previous section (3.1).

Here we proceed to describe the *abstract syntax* for the commands of the Chloe language.

In figure 3.5 we can find the definition of the datatype created in Isabelle for the commands, where `lexp` and `exp` stand for expressions described in the previous sections (3.1), `vname` stands for variable names and `fname` stands for the function names.

For assignment commands we define two different commands, one of them allows assignment to a variable, whereas the other one allows assignment to an

[1] The SKIP command is is equivalent to a noop because it does nothing. We use it in order to be able to express other syntactic constructs such as a conditional without an ELSE branch.

address location in memory. We need these two commands since our domain for addresses and integer values is disjoint, therefore an address cannot represent an integer value and vice versa.

We also have two return commands, one of them is for returning from functions with a return value, whereas the other one is for returning from a function which has no return value.

Finally we have three different statements for function calling. One of them (`Callfunv`) is for functions without a return value. The other two depend on what is done with the return value of the function, if the return value should be assigned to a variable we use the `Callfun` command and if the return is to be assigned to a cell in memory we use the `Callfunl` command.

In Isabelle we have defined a concrete infix syntax as well, which facilitates writing and reading commands in Chloe. In table 3.1 we introduce the concrete syntax supposing we have x that ranges over variable names, a that ranges over expressions, c, c_1 and c_2 that range over commands, y that ranges over the LHS expressions and f that ranges over function names. We will continue to use the concrete syntax throughout the rest of this work to make it more readable.

Abstract syntax	**Concrete syntax**
`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 `(` $[a]$ `)`

Cuadro 3.1: Abstract and concrete syntax equivalence

3.3. Functions

In Chloe we have both functions that return values and those which do not have a return value. For functions that do return a value we needed to figure out what to do with that return value. We decided that the return value of every function should either be assigned to a variable or to a destination in a memory cell or ignored whenever a function was returning from a call. We will not go into details

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

valid_fun_decl :: fun_decl ⇒ bool
```

Figura 3.6: Function definitions

explaining this design decision now but rather delay it until section 3.7.1 where we will be able to explain the reasoning behind this in a better manner.

As we see in the definition in figure 3.6, a function consists of a name, the formal parameters, the local variables and the body of the function, which is a, potentially big, command in the Chloe language. We also define a predicate which checks whether a function declaration is valid or not. A function declaration is considered valid if and only if the function parameters and the local variables have different names.

3.4. Programs

A program in Chloe consists of a name, a list of global variables and a list of functions as showed in figure 3.7. In that same figure we can see a definition that every valid program must comply with.

A program is considered valid if it complies with all the following conditions:

- The names for the global variables are different from one another.
- The names for the functions in the program are different from one another.
- Every function declaration for every function in the program must be valid.
- The main function must be defined.
- None of the variable names or function names in the program must be a reserved keyword from C or a reserved keyword for testing.[2]
- The global variables and the function names in a program cannot be the same.

[2] Since we want to generate C code from the Chloe semantics we must guarantee that neither variable nor function names are any of the reserved C keywords or any of the reserved keywords used for testing variables.

```
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)))
```

Figura 3.7: Program definitions

3.5. Restrictions

This semantics does an assumption regarding the machine where the code is going to be executed. This restriction is parameterized and corresponds to the architecture of the machine where our code is going to be ultimately executed. As stated earlier, the precision of the integer values can be changed in order to make this semantics compatible with different architectures, e.g. 32-bit architectures.

We can change the assumptions this semantics makes by changing the `int_width` parameter, which will automatically change the upper and lower bounds we assume for integers (these bounds are described in figure 3.2).

In order to guarantee the assumptions our semantics makes, later in the code generation process, we generate assertions that make sure the conditions we assume are met.

Another restriction in our semantics is that it only works for a subset of C where the semantics is deterministic and we consider any undefined behavior to be an error in our semantics. We will go to an erroneous state if our semantics encounters behavior that is undefined by the C standard[ISO07], an example of undefined behavior is integer overflow.

3.6. States

In simpler languages, which only support a limited set of features such as assignment, sequential composition, conditionals, loops and integer values, the state representation consists of simply a function that maps variables to values. In Chloe that is not the case, by including constructs as functions, dynamic memory and pointers to our set of features, the state representation ceases to be a simple function that maps variable names to values. In this section we will detail the complete components of the representation of a state. Previously, in section 3.1.2 we began explaining a simplified “visible state”, here we will clarify the difference between that visible state and a real state, as well as detailing the components of a state in a program.

3.6.1. Valuation

As mentioned previously a valuation is simply a function that maps variable names to values. The return type for this function is `val option option` which models a variable having one of the following states: undefined, uninitialized or holding a value.

3.6.2. Stack

Chloe supports functions calls, in order to do so an execution stack must be maintained in the state. This execution stack (from now on we will refer to it simply as the stack) consists of a list of stack frames. Each of these stack frames contains important information about the current function call.

In figure 3.8 the type for a stack frame is a tuple containing a Chloe command, a valuation and a return location. The Chloe command corresponds to the body of the function to be executed. The valuation corresponds to the values of the variables that are local to the function that was called. Finally the return location

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

type_synonym stack_frame = com × valuation × return_loc
```

Figura 3.8: Stack definitions

can be one of the following three: an address, a variable or an invalid return location. When a function returns a variable several things can happen:

- The value will be assigned to a variable. This is indicated by the variable return location
- The value will be assigned to a cell in memory. This is indicated by the address return location
- The value will be ignored. This is indicated by the invalid return location

We will also use the invalid return location for functions that do not have a return value.

The return location is set on the caller's stack frame, that is, when returning from a function the stack frame of the caller is the one to be checked to know where to assign the return value or if a value is expected at all. To further clarify this we can take the example code from figure ??. It is a simple program where a function that adds the value of its two parameters is defined and then called from the main function. Before the function call, the stack frame belonging to the main function has an *Invalid* return location and after the function call, the return location changes to the variable x. Notice that the stack frame that changes its return location is the one belonging to main, this is because the caller is the one saving the return location where it expects to store the value of a returning function. An *Invalid* return value indicates the caller is not expecting to store any results, i.e. either the caller has not called any functions yet or the function it has called have no return value.

3.6.3. Procedure Table

Another extension we must add when dealing with functions is a procedure table. A procedure table is defined as follows:

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

where "⇀ `fun_decl`" is equivalent to writing "⇒ `fun_decl option`". This function maps function names to their declaration.

This function is constructed by taking the program definition and pairing every function declaration from the list of functions to its name. Every program has its own procedure table.

3.6.4. State

A state is defined as a tuple containing the stack, a valuation for the globals and the dynamic memory.

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

3.6.5. Initial State

In order to build an initial state we need to define some components. In figure 3.7 we find definitions for the initial configuration of the stack, the global variables and the memory. The initial configuration for the stack consists of the stack frame for the main function of the program. The initial configuration for the global valuation is a function where every possible variable name is mapped to the undefined variable value (`None`). The initial configuration for the dynamic memory is the empty memory, since nothing has been allocated.

Having defined all of these components the initial state configuration is given by the tuple consisting of the initial configuration for the stack, global variables and dynamic memory.

3.6.6. Visible State

Additionally, a visible state is defined (as mentioned before in section 3.1.2). Executing a transformer function[3] over a state, excepting the transformer functions for function calls or returns, will not be able to modify any part of the stack other than the current frame's local variables valuation. We define the real transformations in the context of visible states and then lift this function to states. Therefore, a transformer function over states, other than the ones for function call or return, cannot tamper with the stack.

The stack frame on the top of the stack corresponds to the current function that is being executed, and the command component of it is the Chloe command

[3] we will cover transformer functions further in section 3.7

```
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: Initial state building

(or program) being executed. Whenever a command is executed or a step is taken in our small-step semantics this command is updated to contain the command that is to be executed next. In order to prove that the small-step semantics is deterministic we must prove that whenever we have a non-empty execution stack the order in which we apply the evaluation transformer[4] and the function that updates the command is irrelevant to the resulting final state. This is why we introduce a separate definition for a visible state apart from the regular state, it is a view of the same state with some limited information.

3.7. Small Step Semantics

3.7.1. CFG

A Control Flow Graph is a graph representation that covers the different paths a program can take during its execution. We have the concept of a current location which is a program pointer to a node. A command can be executed by following edges from the node pointed to by the program pointer to a new node. The nodes in our CFG are commands. The edges in our CFG are annotated with two functions that depend on the current program state. The first one indicates whether an edge can be followed or not (for example, in the case of a conditional) and the second one indicates how the state is transformed by following the edge i.e. the

[4] This is the eval function lifted to work on states instead of visible states, we will discuss it in further detail in section 3.7

effect that following the edge has on the state. We call these two functions *enabled* and *transformer* functions. We will now describe in detail the definitions for these functions.

Enabled functions An enabled function is a partial function as follows:

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

It indicates whether a state is enabled to continue its execution. This is a partial function, therefore its execution might fail. The execution of an enabled function will fail whenever an error is encountered when evaluating the function and it yields a `None` value indicating an erroneous state.

This function is useful for the execution of conditional constructs in our language.

Suppose we have the term "`IF` b `THEN` c_1 `ELSE` c_2". When the evaluation of the condition b yields a `True` value we will follow the edge that leads us to the node that contains c_1. Whereas when the evaluation of b yields a `False` value we will follow the edge that leads us to the node that contains c_2. Depending on the result of the enabled function we will decide whether the term is enabled to follow one edge or the other. The only case where an execution would not be able to continue is when there is no enabled edge to follow. Fortunately, this cannot happen in our programs as there is always an enabled edge. Except for the conditional construct, for every command supported by Chloe the enabled function will always yield `True`. In the case of a conditional construct, the execution can either continue its execution by following the edge to the first command or by following the edge to the second command. There will always be an edge that can be followed after a node that contains a conditional

A list of the enabled functions we use is shown in figure 3.10. The function `truth_value_of` maps a value to a boolean value, namely `True` or `False`. We also find `en_always` which always yields `True`, the `en_pos` function which only yields `True` when the truth value of the expression given as a parameter evaluates to `True` and the `en_neg` function which only yields `True` when the truth value of the expression given as a parameter evaluates to `False`. These functions will be used later in the CFG definition.

Transformer functions A transformer function is a partial function as follows:

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

```
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: Enabled functions

It is a partial function that transforms a state into another. Since it is a partial function, its execution might fail. The execution of a transformer function will fail whenever an error is encountered at some point of its execution and yield a `None` value indicating an erroneous state.

We define functions that will return a transformer function for each command in Chloe. These functions will be used later in the CFG definition and are defined in figure 3.11 We define a transformer function `tr_id` that will serve as the id function and simply yields the same state it is given as a parameter.

The definitions listed in figure 3.11 yield a transformer function which we will use when executing a command. We proceed to roughly describe the effect the transformer functions yielded will have when applied to a state.

First of all we have the transformer for an assignment yielded by `tr_assign`, it will evaluate the expression we want to assign and then proceed to perform a `write` operation to the state and it yields the state resulting from this evaluation and `write` operation.

The transformer for an assignment to a cell in the dynamic memory yielded by `tr_assignl` will first evaluate the LHS expression to obtain the location in memory where the value will be updated, then it will evaluate the expression to obtain the new value and proceed to store the value in memory, it yields the state resulting from the evaluations and the `store` operation.

The transformer for an evaluation yielded by `tr_eval` will evaluate the expression and yield the state resulting from the evaluation.

The transformer for a free operation yielded by `tr_free` will evaluate the LHS expression it receives as a parameter in order to obtain an address and the block for this address will be deallocated. The transformer will yield the state resulting from the evaluation and deallocation of memory.

When performing a function call one must check that the formal parameters and the actual parameters given to the function have the same type and that each formal parameter has a corresponding given parameter. Since we do not have a static type system we will only check the second condition mentioned. Furthermore, we fix the evaluation order for the parameters given to a function as left to right order. The parameters will always be evaluated following a left to right order. Finally when calling a function we must also map the formal parameters to the values of the given parameters and regard these as local variables in the scope of our function.

We have a `call_function` which yields a transformer function for any function call, this transformer checks that the number of formal parameters and given parameters is the same, it evaluates the given parameters from left to right, it creates a new stack frame containing the body of the function, the local variables valuation (which includes the parameters mapped to their given values and the local variables mapped to an uninitialized value) and the return location `Invalid`, it yields the state resulting from performing these operations over the state.

When calling a function the caller has to change its stack frame in order to update the return location value in the current stack frame. We define different functions that perform that change depending on the type of function call and then proceed to yield a transformer by calling `call_function`.

The different functions we define are `tr_callfunl`, `tr_callfun` and `tr_callfunv`. We use `tr_callfunl` when the return location of the function is a cell in memory. In this case the function first evaluates the LHS expression to obtain the return address and updates the return location of the stack frame with it before calling `call_function`, the resulting transformer function will yield a state resulting from this evaluation, the update of the stack frame and the operations done by `call_function`. It is important here that the evaluation of the LHS expression that yields an address value must be the first one done, this is in order to avoid unwanted behavior since the function could change the state.

We use `tr_callfun` when the return location of the function is a variable. In this case the function updates the return location of the stack frame with the variable name before calling `call_function`, the resulting transformer function will yield a state resulting from this update of the stack frame and the operations done by `call_function`.

We use `tr_callfunl` when the function is not expected to return a value. In this case the function updates the return location of the stack frame with

```
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: Transformer functions

`Invalid` before calling `call_function`, the resulting transformer function will yield a state resulting from this update of the stack frame and the operations done by `call_function`.

We have a `tr_return` which yields a transformer function for a return call returning an expression in a function, this transformer will pop the last stack frame in the stack belonging to the returning function, evaluate the value corresponding to the expression returned by the function and if the stack is not empty then proceed to retrieve the return location and depending on if it is an address, a variable or an invalid location it will yield the state resulting from storing the value in memory, writing the value to a variable or yield the state as it is, respectively as well as popping the last stack frame of the stack. Note that if the function returns a value but its return location is expected to be invalid then the returned value is ignored instead of being considered an erroneous execution.

Finally we have a `tr_return_void` which yields a transformer function for a return call in a function which has no return value. This transformer will pop the last stack frame belonging to the returning function. Subsequently, if the stack is not empty it will obtain the return location. If the return location is anything different from an invalid return location it will return a `None` value which represents an error. However, if the return location is, in fact, an invalid one then it will yield the state resulting from the last frame of the stack being popped.

CFG In order to talk about executions by following edges in the CFG we must introduce yet another concept. Taking the definition given at the beginning of this section of what a CFG is we know that in order to talk about an execution of a command by following the edges of the CFG we must have a program pointer that indicates the current node. We also need to introduce the concept of a stack. We will have a stack of program pointers to accompany our CFG and we will pop a

```
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
```

Figura 3.12: CFG rules

new program pointer from the stack once we have `followed` the current pointer up to a `SKIP` node. A program pointer to the command of a function will be pushed to the stack when there is a function call. A program pointer will be popped from the stack when there is a return command. This concept will become important when talking about function calls and return commands further ahead.

We can see an inductive definition in figure 3.12 for CFG, where we can see the rules to form edges between commands. We can follow an edge in the CFG from an assignment to a variable to `SKIP`. This edge is always enabled and has a transformer function to update the state which is given by the result of `tr_assign` called with the parameters specific for the command.

Likewise, an edge in the CFG can be followed from an assignment to a cell in memory to `SKIP`, this edge is always enabled and the transformer function to update the state is given by the result of `tr_assignl` called with the parameter specific for the command.

The sequential composition has two different edges that can be followed. When the first command in the sequential composition is a `SKIP` command we can follow an edge that will lead us from the node with the whole command to a node with only the second command. This edge is always enabled and the transformer over the state is the `tr_id`. The second case occurs when we have a command of the form `(` c_1 `;;` c_2 `)` where c_1 is not `SKIP`. If we follow an edge from c_1 to some other node c_1' (where the edge is labelled a), which contains an enabled and a transformer function, then we can follow an edge (also labelled with a) from `(` c_1 `;;` c_2 `)` to `(` c_1' `;;` c_2 `)`.

We also have two cases when it comes to the case of the conditional. We will have the possibility to follow an edge to the first command or an edge to the second command, depending on the value of the guard. In the case where we follow the edge that leads us to the first command, we will go from `IF` b `THEN` c_1 `ELSE` c_2 to c_1. This edge will be labelled with an enabled function given by `en_pos` which will be enabled only when the condition b evaluates to `True` and with a transformer function given by `tr_eval`. In the case where we follow the edge that leads us to the second command, we will go from `IF` b `THEN` c_1 `ELSE` c_2 to c_2. This edge will be labelled with an enabled function given by `en_neg` which will be enabled only when the condition b evaluates to `False` and with a transformer function given by `tr_eval`.

In the case of a loop we can always follow an edge that takes us from `WHILE` b `DO` c to `IF` b `THEN` c `;; WHILE` b `DO` c `ELSE SKIP` by unrolling the loop once. The edge will have an enabled function that is always enabled and the transformer over the state is the `tr_id`.

We can follow an edge from `FREE` x to `SKIP`. This edge is always enabled and a has transformer function given by the result of `tr_free` called with x.

We can follow an edge from any of the two existing return commands to `SKIP`. This edge is always enabled and has a transformer function given by the result of `tr_return` or `tr_return_void` depending on which return command it is. Note that when *executing* this command in the CFG, following the edge from a return node to a `SKIP` node will pop the program pointer pointing to the `SKIP` node and we will continue our “execution” in the location pointed by the next program pointer in the stack.

Finally, we can follow an edge from any of the function call nodes to `SKIP`. This edge is always enabled and has a transformer function given by the result of either `tr_callfunl`, `tr_callfun` or `tr_callfunv` depending on whether it is a function call that returns to a memory cell, a variable or returns no value. Note that, also in this case, when *executing* this command in the CFG, following the edge from a function call node to a `SKIP` node will push a new program pointer to the stack that points to the location of the node that contains the command of the function and we will continue our “execution” in the location pointed by this next program pointer.

3.7.2. Small Step semantics rules

Finally, we introduce the rules for the small-step semantics for Chloe. A small-step semantics is chosen over a big-step definition for the semantics due to the fact that we want a finer grained semantics. The big-step semantics has a major drawback, which is that it cannot differentiate between a non-terminating execution and getting stuck in an erroneous configuration. This is why we prefer a more detailed semantics that allows us to differentiate between nontermination and getting stuck because it allows us to talk about intermediate states during evaluation.

Usually a configuration in a small step semantics is a pair consisting of a command and a state. Since we are working with functions and we have the command that is being executed in the stack frame our small-step definition is defined over states. A small, atomic step can be taken from one state to another.

The small-step rules for the semantics are detailed in figure 3.13. The infix syntax for the small step semantics is written as $s \rightarrow s_2$ which means we take a small step from s to s_2. A step can be taken if the following conditions are met:

- The stack is not empty.
- There is a CFG edge between c_1 and c_2.
- The command at the topmost stack frame in the initial state is c_1.
- Applying the enabled function over the state yields `True`.
- Applying the transformer function to the state with the command at the topmost stack frame updated from c_1 to c_2 yields a new state s_2.

Given that all the previous conditions are fulfilled then a small step can go from state s to s_2.

A small step can also be taken upon return from a function which returns no value. If the command at the top of the stack is `SKIP`, the stack is not empty and applying the transformer function on the initial state s yields a new state s_2, then we can take a small step from s to s_2

The type for `small_step` is from `state` to `state option`. The second configuration is enclosed in an option type because taking a small step can result in an erroneous state in which we will get stuck.

A small step can fail to be taken in any of the following cases:

```
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 3.13: Small-step rules

- Either the enabled function or the transformer function yield `None` when evaluated over the initial state. This indicates an erroneous state was reached when evaluating one of those functions. We propagate the erroneous state by taking a small step from state s to `None` and then the execution will get stuck there.

- If applying the transformer function `tr_return_void` over the state yields a `None` value, the command at the top of the stack in the initial state is `SKIP` and the stack is not empty, then we also propagate this erroneous `None` state by taking a small step from s to None. This indicates there was an error returning from a function without a return value.

We have defined how to take a single step in our semantics. In order to take more than one step and define the execution of a program in our semantics we must lift the definition of `small_step` to `state option` in both the initial and the final state.

We also define (in figure 3.13) a new `small_step'` based on our previous definition of `small_step`. This definition essentially says that if a small step can be taken from state s to state s' then a small step can be taken from `Some` s to s'. Note that s' can either be a `None` value or a `Some` s_i. The infix syntax for this new `small_step'` is written as `Some` $s \rightarrow' s'$, which means we take a small step from `Some` s to s'.

In this manner, we have lifted our small step definition to the type `state option` and we can define the execution of a program as the reflexive transitive closure of the newly defined `small_step'`, using Isabelle's `star` operator. The infix syntax for this is written as $s_0 \to *s_f$, which means we can go from state s_0 to s_f in zero or more small steps.

Determinism

To be able to make our semantics executable inside of the Isabelle/HOL environment it must be deterministic. This is why we prove the determinism property for the semantics. In order to go through with that proof a set of lemmas must be first defined and proved. We will proceed to explain such defined lemmas for our semantics in this section.

Whenever we introduce a lemma, next to its number we will have a name in parenthesis. This corresponds to the name of the lemma in the Isabelle source files submitted with this work. In the case where the reader is interested in the exact proof for a particular lemma he or she can search for the corresponding lemma in the theory files and read the proof.

This first lemma indicates that whenever we have a command different from `SKIP` there is either always an enabled action to take in the CFG or an error occurs when trying to evaluate the enabled function.

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

Demostración. The proof is by induction on the command. Except for two cases the proof is solved automatically. Those interesting cases are the `Seq` and the `If` cases. In the `Seq` case we need to make a further case distinction on the first command of the sequential composition in order to differentiate the cases where it is `SKIP` and when it is not. In both cases there is a CFG rule that ensures there is an enabled action and the case is solved automatically.

In the `If` case we need to make a case distinction over the value of `en_pos` $b\ s$. It can fail and return a None value, then the case is solved. If it does not fail then we must check the boolean value returned by the `en_pos` function. In the case where it is `True` then the case is solved. The challenging case comes when the evaluation of `en_pos` is `False`, semantically, this means the else branch should be taken instead. Therefore when `en_pos` $b\ s$ is `False`, `en_neg` $b\ s$ will always evaluate to `True`, this means that the command will have an enabled action, namely `en_neg` $b\ s$ and the proof of this case is complete. □

Next we want to prove that as long as the stack is not empty the small-step semantics can always take a step. We will make use of the previous lemma in the proof for this new lemma. This lemma states that the semantics can take a step.

Lema 3.2 (can_take_step).
$\neg$ `is_empty_stack` $s \Longrightarrow \exists\, x.\ s\ \rightarrow\ x$

Demostración. From the assumptions we know that the stack is not empty, therefore we can rewrite the state as follows: $s = ((c, locals, rloc)\#\sigma, \gamma, \mu)$. We can then do a proof by cases. We have the case where $c\ =$ `SKIP` and the case where c *neq* `SKIP`.

The case where $c\ =$ `SKIP` is the case where we are returning from the execution of a function. We know that the stack is not empty, that $c\ =$ `SKIP` and that $s = ((c, locals, rloc)\#\sigma, \gamma, \mu)$. This case corresponds to the `Return_void` and `Return_void_None` rules in the definition of `small_step`. In both these cases the semantics can take a step, either to some new state s' or to `None`. This goal is solved automatically by Isabelle indicating the mentioned rules and assumptions.

The second case is where c
neq `SKIP`. This is the case when we are executing any other command and not returning from a function. In this case we use the previous lemma 3.1 and from that we know that `cfg` c (en, tr) c' and either the enabled function fails (`en` $s\ =$ `None`) or it is enabled (`en`| $s\ =$ `SomeTrue`).

We prove this subgoal by splitting it into cases. In the first case we assume `cfg` c (en, tr) c' and `en` $s\ =$ `None`. This is the case where the evaluation function fails, we know this case takes a step to None because of the `None` case of the small step definition. Then we have proved that it exists a state so that $s\ \rightarrow$ `None`.

In the next case we assume `cfg` c (en, tr) c' and `en` $s\ =$ `SomeTrue`. This is the case where we are enabled to take a step and we must still check two more cases. Applying the transformer function over the state updated with the command (tr (upd_com c' s)) can either fail or not. In the case it fails (it returns `None`) we can take a step to `None` and we will have proved that there is a state to which s can take a small step, namely `None`. In the case it does not fail, it will return `Some` s_2. In this case we can take a step to `Some` s_2 and we will have proved that there is a state to which s can take a small step, namely `Some` s_2. □

We define several lemmas that will come in handy later on.

The first lemma states that the CFG gets stuck at SKIP:

Lema 3.3 (cfg_SKIP_stuck).

$\neg\ \texttt{cfg SKIP}\ a\ c$

Demostración. The property is proved automatically. □

Lema 3.4 (ss_empty_stack_stuck).

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

Demostración. The property is proved automatically. □

Lema 3.5 (ss'_SKIP_stuck).

$\texttt{is\_empty\_stack}\ s \Longrightarrow \neg\ (Somes\ \rightarrow\ cs')$

Demostración. The property is proved automatically. □

Lemmas 3.4 and 3.5 define the final state in which the semantics will get stuck, the semantics will get stuck when there are no more stack frames in the stack.

Lema 3.6 (en_neg_by_pos).

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

Demostración. The property is proved automatically unfolding the definitions of `en_neg` and `en_pos`. □

Lemma 3.6 states that every time the enabled function `en_neg` has a value (different than None) over a state, `en_pos` will have the same but opposite result. In the case where one of the functions fails the other one will too, and they will yield `None`. If they do not fail their results will be opposite, that is, if one has `Some True` as a result, the other one will have `Some False` as a result. This lemma will come in handy when proving determinism.

Lema 3.7 (cfg_determ).

$\texttt{cfg}\ c\ a1\ c' \wedge \texttt{cfg}\ c\ a2\ c''$

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

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

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

Demostración. The proof is by induction on the command, with the cases of the `cfg` rules generated by Isabelle all cases are solved automatically. □

Lemma 3.7 states that CFG is deterministic. The only case where it is not is in the conditional case, for which we add an extra alternative in the conclusion. It can happen that we have a CFG rule starting at an `If` command that has an edge to a c_1 and also an edge to c_2. This is not really a problem since the enabled functions guarantee that whenever that happens only one of the branches can be taken.

Lema 3.8 (lift_upd_com)**.**
$\neg$ `is_empty_stack` $s \implies$
`lift_transformer_nr` tr (`upd_com` c s) $=$
`map_option` (`upd_com` c) (`lift_transformer_nr` tr s)

Demostración. It is proved automatically by unfolding the definition of `lift_transformer_nr`. □

Lema 3.9 (tr_eval_upd_com)**.**
$\neg$ `is_empty_stack` $s \implies$
`tr_eval` e (`upd_com` c s) $=$
`map_option` (`upd_com` c) (`tr_eval` e s)

Demostración. It is proved automatically by unfolding the definition of `tr_eval`. □

The `lift_transformer_nr` function lifts to states the definition of a transformer function that operates in the visible state level. Lemmas 3.8 states that it does not matter in which order a transformer function is applied over a state and the command in the top of the stack frame is updated, since it will always yield the same result. This is because the update command function only modifies the command at the top of the stack and the transformer function cannot access and modify that part of the stack since it only modifies the visible state.

Lemma 3.9 is a more specific version of the previous lemma that states the same fact specifically for the evaluation transformer function.

All the previous lemmas and definitions have been building up to this next proof, the determinism of the small step semantics.

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

```
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 3.14: Single step edges

Demostración. The proof is by cases on the small step semantics. We obtain 4 cases, each corresponding to each rule in the small step semantics. The goals generated by the `Return_void` and `Return_void_None` rules are solved automatically. The goals generated by the `Base` and the `None` are solved automatically after adding lemmas 3.6 and 3.9. □

Then we are only left to show that `small_step'` is also deterministic.

Lema 3.11 (small_step'_determ).
$s \to' s' \wedge s \to' s'' \Longrightarrow s' = s''$

Demostración. The proof is by cases on the small step semantics. It is proved automatically by using lemma 3.10. □

3.8. Interpreter

3.8.1. Single step

There are two kinds of edges in the CFG. We create a new datatype for representing them.

```
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 3.15: Definition of fstep

A *Base* edge that has a transformer function and is always enabled and takes us to a new command and a *Cond* edge that, apart from the transformer function, also has an enabled function and two commands. This enabled function indicates whether we take a step to the first or the second command. Also we define a `cfg_step` function that given the starting command returns which kind of edge follows in the CFG.

The function `fstep`, defined in figure 3.15, is how we take a single step in the execution of the semantics. An execution of a step in the semantics will take us from an initial state to a new state which can be an erroneous state (`None`) or a valid one (`Some` s). To execute a `SKIP` command we call the transformer function for returning without a value from a function `tr_return_void`. Otherwise we will check which kind of step we should take by means of the `cfg_Step` function and, based on that, decide what to do. If it is a `Base` step we will call the transformer function over the state with the updated command. If, on the other hand, it is a `Cond` step we will evaluate the condition and call the transformer function over the state with the updated command.

Equality between small-step semantics and single-step execution

We must now prove that the single step execution is semantically equivalent to taking a step in the small-step semantics. This is proving that $\neg$ `is_empty_stack` $\Longrightarrow s \rightarrow s' \longleftrightarrow$ `fstep` $s = s'$[5]. We prove both directions of the equivalence separately: for every step taken in the small-step semantics, there is an equivalent

[5] fstep has an extra parameter, namely the procedure table, which we will avoid writing here to make it simpler to read

```
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 3.16: Definition of an interpreter for Chloe

step that can be taken in the execution by `fstep` which will lead to the same final state and vice versa.

We start by showing that any step taken in the small-step semantics can be simulated by a step taken with fstep.

Lema 3.12 (fstep1)**.**
$s \rightarrow s' \Longrightarrow \texttt{fstep}\ s = s'$

Demostración. This proof is done by induction over the small-step semantics. □

Then we consider the other direction:

Lema 3.13 (fstep2)**.**
$\neg\ \texttt{is\_empty\_stack}\ s \Longrightarrow s \rightarrow (\texttt{fstep}\ s)$

Demostración. This proof is done automatically by making a case distinction on the result of "`tr_return_void` s" and using the lemmas 3.2 and 3.12. □

Both directions together (Lemma 3.12 and lemma 3.13) let us then show the equivalence we were aiming for originally:

Lema 3.14 (ss_fstep_equiv)**.**
$\neg\ \texttt{is\_empty\_stack} \Longrightarrow s \rightarrow s' \longleftrightarrow \texttt{fstep}\ s = s'$

3.8.2. Execution and Interpretation

In order to execute a program we will define an interpreter for it.

In figure 3.16 we find such definition. First we define the criteria on which a state is considered final. A state will be considered final when its execution stack is empty or when it is `None`.

The interpreter for our semantics works as follows: As long as a final state is not reached we execute `fstep`.

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

Figura 3.17: Definition of an interpreter for Chloe

Finally, we show a lemma that states that if a state is final, then it is the result of the interpretation.

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

In order to show this we need a lemma that unfolds the loop in the definition of our interpreter:

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 $\})$

Demostración. The proof is solved automatically. □

With lemma 3.16 the proof for lemma 3.15 is solved automatically.

Finally, only valid programs can be executed. In figure 3.17 we can see the definition for the function that executes a program. In order to execute a program we assert that it is valid with the previously defined `valid_program` and then proceed to interpret the initial state of the program p.

3.8.3. Correctness

Finally we must show our interpreter to be correct. In figure 3.18 we have two definitions regarding execution. The first one states that an execution of a state cs `yields` cs' if we can take small steps from cs to cs' and cs' is a final state. Secondly, we define an execution of a state to terminate if there exists a state cs' such that the execution of state cs yields it.

Before showing the correctness lemma for the correctness of the interpreter we must show that the small steps execution will preserve an erroneous `None` state if it is reached by the path of steps taken. Upon erroneous execution we will get stuck in a `None` state.

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

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

FIGURA 3.18: Definitions about program execution

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

Demostración. The proof is by induction on the reflexive transitive closure (star). The goals are solved automatically. □

Finally we show the correctness property for our interpreter. Theorem 3.18 states that if the execution of cs terminates, then that execution yields cs' if and only if cs' is the result we obtain from executing the program in the interpreter.

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

Demostración. The proof is done by assuming the premise and proving each direction of the equality separately. □

Capítulo 4

Pretty Printer

In this chapter we will detail the *pretty printing* (translation) process that happens in the semantics and allows us to export C code. To assist us in this translation process we used Sternagel and Thiemann's implementation of Haskell's Show class in Isabelle/HOL[ST14]. Sternagel and Thiemann implement a type class for "to-string" functions as well as instantiations for Isabelle/HOL's standard types. Moreover they allow for deriving show functions for arbitrary user defined datatypes. We instantiate this class creating a "show" function for each of our defined datatypes progressively until we can print a program. The following sections will explain in detail the concrete strings we obtain as a result of our translation.

4.1. Words

The first instantiation of shows we must make is the one for words in order to be able to pretty print values of this type. For pretty printing a value of the type word we will simply cast that value to an Isabelle/HOL's predefined int type and use the `show` function for it. As a result our words will be pretty printed as signed integers.

4.2. Values

Although the datatype `val` and valuations are not used in the code generation process, we find it useful to have a mechanism to pretty print them for debugging purposes.

4.2.1. Val type

The type `val` is the first user defined type we provide an instantiation for. In table 4.1 we find the equivalence between the abstract syntax for the `val` type and the string representation. Note that w, *base* and ofs range over words, nats and integers, therefore their show functions will be used to obtain their string representation e.g. the string representation for `I` 42 would be ”42”, the string representation for `A` (4, 2) would be ”4[2]”. A list of values is showed by showing each value in a list notation, i.e. $[vname = value, \ldots, vname_n = value_n]$.

We will enclose parameters in between brackets (” $<>$ ”) to indicate that what's enclosed in them is a string representation which results from applying a shows function on the parameter and will continue to use this notation throughout the rest of the document.

Abstract syntax	**String representation**
`NullVal`	null
`I` w	$< w >$
`A` ($base$, ofs)	¡$base$¿[¡ofs¿]

Cuadro 4.1: Translation of val type

4.2.2. Val option type

Now that we know how to represent a `val` type in the form of a string we must also know how to represent a `val option`. A `val option` holds the semantic meaning of an initialized and an uninitialized value. Table 4.2 shows the equivalence between the abstract syntax and the string representation. Here an initialized value will simply be the string resulting from showing that value, whereas the string representation of an uninitialized value will be a ″?″ to represent its value is not yet known.

Abstract syntax	**String representation**
`Some` v	¡v¿
`None`	?

Cuadro 4.2: Translation of val option type

4.2.3. Valuations

It must also be possible to have a string representation of a valuation. In order to represent a valuation we need an extra parameter, namely a list of variable names, this list will have the names of the variables for which we want to print their value in the valuation. The valuation will be printed in the following format:

$$[< vname_0 >=< value_0 >, < vname_1 >=< value_1 >, \ldots, < vname_n >=< value_n >]$$

For instance, if we take the valuation $[foo \mapsto \texttt{Some}(\texttt{I}\ 15), bar \mapsto None, baz \mapsto \texttt{Some}(\texttt{A}(1,9))]$ and the list of variables $[foo, bar, baz]$ then we would obtain the following as a string representation for the valuation:

$$[foo\ =\ 15,\ bar\ =?,\ baz\ =\ 1[9]]$$

4.3. Memory

The memory, same as the values, is not a component we need in order to generate C programs. Nevertheless, having a method to pretty print the memory in a determined state is useful in the event of debugging. In order to show the memory we must first know how to show a string representation of the content of a block and a string representation of the whole block.

When we access a block in the memory we can obtain either the content of the block or a `None` value indicating a free block of memory. Table 4.3 shows the string representations for this. Notice that in the case of a `None` value we print the `free` string enclosed in brackets, being the brackets part of the string representation and posing an exception to the notation described previously. If the content of the block is a list of values then we show the list of values.

Abstract syntax	**String representation**
`Some` content	¡*content*¿
`None`	¡free¿

Cuadro 4.3: Translation of Block content

A complete block will be printed as follows:

$$< base > : < block\_content >$$

where *base* is the first component of an address value that indexes the blocks and $block_content$ is the string representation of the content of block number *base*.

Finally, in order to show the whole memory, we only need to show each block existing in the memory. For instance, the string representation of memory state:

$$[\texttt{Some}\ [\texttt{I}\ 13,\ \texttt{None}],\ None,\ \texttt{Some}\ [\texttt{A}(2,3),\ None, \texttt{I}\ 56]]$$

would be:

$$\begin{aligned} 0 &: [13,\ ?] \\ 1 &: < free > \\ 2 &: [2[3],\ ?,\ 56] \end{aligned}$$

4.4. Expressions

In order to print expressions we must be able to print unary and binary operations. We must also be able to print casts from and to pointers. We use C's `intptr_t` which is big enough to hold the value of an integer as well as the value of a pointer, we do so due to the fact that we only have integer values in our language and we separate address values from them in the semantics level.

At the C level we must be able to tell the compiler when some value is meant to be an address and cast it as such. We allow this casting between addresses and integers during the translation process because we know for sure when a value should be interpreted as an address and when it should be interpreted as an integer, whereas C does not.

Unary and binary operations When pretty printing binary operations we will use parenthesis around every expression pretty printed. This will naturally generate more parenthesis than needed but we are willing to make this choice to ensure the evaluation order remains the same as intended and we do not obtain different evaluation orders because of operator precedence.

A binary operator is pretty printed in an infix way:

$$(< operand_1 >< operator >< operand_2 >)$$

Examples of this are shown in table 4.4.

Abstract syntax	String representation
`Plus (Const 11) (Const 11)`	(11 + 11)
`Subst (Const 9) (Const 5)`	(9 − 5)
`Mult (Const 2) (Const 3)`	(2 ∗ 3)

Cuadro 4.4: Examples of binary operators' pretty printing

An unary operator is pretty printed in a prefix way:

$$< operator > (< operand >)$$

Notice we enclose the operand in parenthesis in order to guarantee correct precedence in the operations.

Examples of this are shown in table 4.5.

Abstract syntax	String representation
`Minus (Const 11)`	− (11)
`Not (Const 0)`	! (0)

Cuadro 4.5: Examples of unary operators' pretty printing

Casts Since the values we are working with must be interpreted in C sometimes as integers and sometimes as pointers we must be able to pretty print an explicit cast between those two types in our generated program. We include casts to pointers when dealing with referencing, dereferencing and indexing. We will want to cast to integers in the case of a memory allocation. A memory allocation returns a pointer but in order to assign that to a variable we must cast it to an integer. Due to the fact that all our variables are declared with the `intptr_t` type and not `intptr_t *`, we can do this and we know that when working with addresses these will be interpreted the right way since we will add a cast back to pointer.

A cast to an address value will be pretty printed in the following way:

$$(\texttt{intptr\_t} \ast) < expression >$$

A cast to an integer value will be pretty printed in the following way:

$$(\texttt{intptr\_t}) < expression >$$

In table 4.6 we can find examples of the pretty printing of casts.

Abstract syntax	String representation
Deref (V *foo*)	*((intptr_t *) foo)
Ref (V *foo*)	((intptr_t *) &(foo))
New (Const 9)	(intptr_t) __MALLOC(sizeof(intptr_t) * (9))

Cuadro 4.6: Examples of casts' pretty printing

Memory allocations As we mentioned in chapter 3 the behavior of our allocation function and C's allocation function differ due to the fact that we assume that the memory is unlimited. Therefore we cannot simply translate our memory allocation function to a `malloc` call in C.

We must wrap C's `malloc` function in another function that will abort the program in the case a program runs out of memory. We define a function "`__MALLOC`" that takes the size of the new block of memory to be allocated with `malloc`, does the malloc call and returns the pointer to the new block of memory in case it succeeds and upon failure of the `malloc` function it aborts the program with the exit code 3. We define exit code 3 as an erroneous exit code that means there was a failure when allocating memory in order to be able to catch this error later in the testing process.

Expressions Finally we present in table 4.7 the string representation for each of the expressions. We use simple expressions as operands such as variables or constant values for simplicity, but the expressions can be made up of more complicated expressions.

4.5. Commands

First, we need a method for printing indented commands to facilitate the generated code. We define two auxiliary abbreviations that will pretty print white spaces for indentation at the beginning of a construct. The reason why two abbreviations are defined is because one of them will also pretty print a ";" terminator after the construct whereas the other will not.

We also define a way of pretty printing function calls. Function calls will be pretty printed according to the following format:

$$< function\_name > \ \ ([< argument_0, argument_1, \ldots, < argument_n >])$$

Abstract syntax	**String representation**
Const 42	42
Null	(intptr_t *) 0
V x	x
Plus (Const 2) (Const5)	(2 + 5)
Subst (Const 9 (Const5)	(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])

Cuadro 4.7: Examples of Expressions' pretty printing

where the brackets ([]) indicate that the arguments are optional.

Finally the commands in Chloe are pretty printed as the examples in table 4.8 shows. We use $''$ to indicate an empty string. The correct level of indentation is indicated in the function parameters that are responsible for the pretty printing, we will omit those here and instead list where the indentation will increase. Indentation will increase when printing commands that are within a block, e.g. the branches of a conditional, the body of a loop.

Abstract syntax	String representation
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 ([])	$bar();$

Cuadro 4.8: Examples of Commands' pretty printing

4.6. Function declarations

Now we must define how declarations are pretty printed. In order to do so, we will pretty print a function definition according to the following string format:

intptr_t $< function\_name >$ (intptr_t $< arg\_name_0 >$, $\ldots$, intptr_t $< arg\_name_n$) {

intptr_t $< local\_var_0 >$

$\vdots$

intptr_t $< local\_var_n >$

$< body >$

}

The return, argument and local variable's type is `intptr_t` since, as mentioned previously, we only have one type in our translation process and we cast to and from pointers when necessary.

An example of a function declaration translation for a factorial function is available in table 4.9. In this example we avoid the use of $''\ ''$ for string representations in Isabelle and simply write the string without the quotation marks.

Abstract syntax	**String representation**

`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);`
	`}`

Cuadro 4.9: Pretty printing of a factorial function declaration

4.7. States

When executing a program inside the Isabelle/HOL environment we will often want to inspect the states. In this section we define an easy way to inspect the states by having a string representation for them. That is precisely what we define in this section.

First we describe how a return location is pretty printed. We instantiate the show class for the type `return_loc`. In table 4.10 we find the equivalence between the abstract syntax for the `return_loc` type and the string representation. Where $< base >$ and $< ofs >$ are the result of applying the show function over *base* and ofs and $< invalid >$ is a literal string including the arrow heads.

Now we describe how the stack is pretty printed. A single stack frame is pretty printed by printing the command, the list of local variables and the return location expected with the following format: $rloc\ = < rloc >$ separated by a line break. In order to pretty print the whole stack, we will print each stack frame separated by “`---------------`”.

Abstract syntax	String representation
`Ar` (*base*, *ofs*)	¡*base*¿[¡*ofs*¿]
`Vr` *w*	*w*
`Invalid`	¡invalid¿

Cuadro 4.10: Translation of return location type

In order to pretty print a state we must give the show function a list of variable names, which will be used to print the valuation for the globals and the locals in the stack frame. A state is pretty printed by printing the stack, the values of the global variables and finally the memory, separated by “================”

An example of pretty printing for a simple state is shown in table 4.11.

Abstract syntax	String representation

([(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]`

Cuadro 4.11: Pretty printing example of a state

4.8. Programs

In this section we can now talk about translating a complete program.

Header files and bound checks In the exported C code we will want to include the header files for the C standard libraries (`stdlib.h` and `stdio.h`). Also we include two more header files that allow the use of `intptr_t` type and a header where the macro definitions that specify limits of integer types are defined. These are `limits.h` and `stdint.h`, respectively. We also want to include the header file that defines some macros used by our test harness for testing purposes, this will be addressed in more detail in chapter 5. Finally we include the header file that contains the function for handling malloc calls. Before pretty printing any part of our program we will want to pretty print the directive to include the header files mentioned before.

Additionally, our translation process generates a program that will be compilable and executable. This is true for architectures that support at least the same range of integer types as our abstraction does. This means that, any architecture where the program will be compiled and executed must comply with the restrictions for integer values that we assume. Therefore, we use C's preprocessor to our advantage and pretty print an integer bounds check that will assert that the macros with the lower and upper bound definition are in fact defined. In our case we must check that the macros `INTPTR_MIN` and `INTPTR_MAX` are both defined. Next, we proceed to assert that the bounds defined in those macros are in fact the same as the bounds we assume (the bounds we assume are `INT_MIN` and `INT_MAX` and they are defined in figure 3.2).

The type we use for our translation as well as the precision of the integers can be changed. We define values and proceed to use them in the semantics and the pretty printing process. The definitions for the type used for translation are as follows:

```
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''"
```

where @ stands for the append operation between strings. It is important to note that the precision of the type used as `dflt_type` must match the precision of `int_width`.

Global variables The only other thing we must know how to pretty print before defining the string representation of a program is a global variable definition. It is done the same way as the pretty printing of local variables, following this string format:

```
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]
    )"
```

Figura 4.1: Factorial definition in Isabelle

$$\begin{array}{l}
\texttt{intptr\_t} \;\; < \textit{variable\_name}_0 >; \\
\vdots \\
\texttt{intptr\_t} \;\; < \textit{variable\_name}_n >;
\end{array}$$

Program The pretty printing of a program is done by printing the include directives of the header files, then the integer bound checks, the global variable declarations and finally, we pretty print each function in the program separated by a line break character. In figures 4.1 we can find the Isabelle definition of a factorial program and in figure 4.2 we find the generated C code for it.

The integer bound check for the lower bound has a workaround because the absolute value of `INTPTR_MIN` overflows the upper bound for integers of the preprocessor and causes a warning.[1] To eliminate that warning we instead compare the `INTPTR_MIN + 1` value to `INT_MIN + 1`.

[1] It interprets the negative number as $-(number)$ and yields a warning that it cannot represent the *number*.

```
#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));
}
```

FIGURA 4.2: Translated C program

4.9. Exporting C code

Now that we know how to translate a complete program from the semantics to C code we are almost ready to export our generated C code.

We only export code for valid programs. To guarantee this we define a function that prepares a program for exporting:

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

This function takes a program and asserts that it is valid, according to the definition of a valid program given in 3.4. If it is not valid we return a `None` value, if it is valid we proceed to generate the string containing the C program with the function `shows_prog`[2].

We export C code to external files. The name of the files is given by the program name in its definitions. We define an ML function inside Isabelle that exports the C code of a program:

```
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)
```

The first parameter of the function `export_c_code` is a `string option` this corresponds to the string representation of the program in C code. If it receives a `None` value this means the `prepare_export` function failed and we do not want to generate a C program for it. The second parameter is the path to the directory where we want the program to be exported, this parameter is a relative path to the directory where the theory containing the pretty printing directives is. The third parameter is the name of the program. The last parameter is the theory context we are in.

If the path given is an empty string the generated C code will be pretty printed to Isabelle's output view. This function will create a new file with the name indicated in the parameters with an extra ”`.c`”(i.e. < *name* > `.c`) added in the directory indicated by the path parameter. The user will then be able to find the generated code in the directory indicated.

[2] We do not include the code for the show functions in this document but rather explain how they work. The Isabelle theories can be checked for implementation details

We also have defined a function in ML that we use when we write an erroneous program on purpose and we expect its execution and translation to C fail:

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

The function `expect_failed_export` will then generate an error in Isabelle if code is generated for the program when we expect it to fail and it will do nothing when no code is generated. Having this kind of function is useful later for testing purposes.

Capítulo 5

Testing

In chapters 3 and 4, we have formalized the semantics for Chloe and described the translation process to C code. Now we proceed to describe the testing process done to increase the trust in our translation process. We guarantee that the code generated from our semantics will either end with an out of memory error or it will yield the same result as the same program executed in our semantics. We say that the result of an execution of the semantics and the execution of the generated program in the machine is the same if the final states yielded by both are the same. This means that we compute the final state yielded by our semantics and verify that after executing the generated program the state is the same i.e. the contents of the reachable memory and the global variables are the same. When generating programs we can either simply generate the C code by itself or we can include a set of extra tests in the form of C macros that will guarantee what we mentioned before.

In the following sections we will proceed to describe the test harnesses used to generate tests for our code. We will also give a more detailed description of the describe the meaning of the two final states being the same. Finally, we will talk about a set of tests and example programs written in the semantics. We only generate code for valid programs. This means that we will go to an erroneous state if any undefined behavior arises. In this set of tests we include programs which we expect to reach an erroneous state because they present undefined behavior or border cases. For these *incorrect* programs C code will not be generated. We also present some example programs such as sorting algorithms to demonstrate how our semantics and code generation process work.

5.1. Equality of final states

We consider a final state yielded by the execution of our semantics equal to a final state of its generated C program, when, at the end of the execution, the values of the global variables are the same for both cases and every block of reachable memory has the same content. We will now proceed to describe how we check for this equality between final states by the use of tests.

5.1.1. Generation of Tests

We can generate tests for our programs. These will be executed at the end of the execution of the program and will test that the final state of the generated program is the same as the final state from the execution of the semantics. We compare these final states by checking the values of global variables at the end of an execution against the ones in the final state of the execution of the semantics.

The direction in which the testing is done is by taking the values from the final state of the execution of the semantics and checking whether the execution of the generated code has the same values we expect it to have according to the execution of the semantics.

We will now introduce which kind of tests are done depending on whether the content of a global variable is an integer value or a pointer value.

5.1.1.1. Integer Values

When the content of a global variable we want to check is an integer value, we must simply generate a test that will check whether the integer value in the global variable at the end of the execution of the C code is the same value as the one we get from the global variables valuation in the final state of the execution of the semantics.

5.1.1.2. Pointers

With the checks for pointers we have two cases. We have the null pointer and the non-null pointer case.

For null values we will generate a test, similar to the one for integer values, where we check if the content of the global variable is `NULL`.

In the case of pointer values that are different from null, we will have a pointer to a block in the memory and we want to check if the content of that block in memory, at the end of the execution of the generated program, is the equal content of that same block in our final state in the semantics. That complete block qualifies as reachable memory which is why we must check the content of each cell in the block. For each cell in the block we will generate checks depending on what the expected content in the memory cell is, i.e an integer value, a null value or a pointer.

In the case of the pointer value checks, we will follow every pointer until we either reach an integer value or a pointer we already followed. Upon reaching an integer value, we will generate an integer kind of test and upon reaching a pointer we already followed, we know that the path does not contain any pointer to invalid memory. We check that the address for the beginning of the block is the same as the one we obtain from adjusting the pointer we followed to the beginning of the block. In order to do this, we must follow pointers in a certain order and maintain a set of already *discovered* blocks of memory. This way when we find a pointer to a block of memory we already checked (or *discovered*) we can stop and compare the pointer values instead of following the pointers in a cyclic manner indefinitely.

We present here the intuitive idea behind our tests generation and in the following sections we will describe the implementation details for the test harnesses, both in Isabelle and in C.

5.2. Test Harness in Isabelle

In this section we will introduce the Isabelle test harness that assists us in the generation of tests for our programs. First, we define a new datatype for every kind of test instruction we can generate. We can see this definition in figure 5.1.

We have four different test instructions we can generate:

- `Discover` represents an instruction that adds a block to the list of our *discovered* blocks. The `string` stands for the string representation of the expression in C and the `nat` stands for the identification number of the current memory block. The actual addresses for the allocated memory blocks will vary with every execution of the program in the machine. The discover instruction pairs the actual address of a beginning of a block with the base block number to which it corresponds in our abstract representation. For this purpose we generate local variables with the function `base_var_name` that are called `__test_harness_x_`*n* where *n* represents the identification

```
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: Definitions for the test harness

number for the block and in those variables we will save the actual address for the beginning of the block for that particular execution.

- `Assert_Eq` represents an instruction that will check that the value to which an expression evaluates is the same as the integer value we expect it to have. The `string` stands for the string representation in C of the expression and the `int_val` stands for the value we expect that variable to have according to our final state in the semantics execution.

- `Assert_Eq_Null` represents an instruction that will check that the value to which an expression evaluates is the null pointer. The `string` stands for the string representation in C of the expression.

- `Assert_Eq_Pointer` represents an instruction that will check that the pointer value to which an expression evaluates points to the same block we expect it to point. The `string` stands for the string representation in C of the expression and the `nat` stands for the identification number for the block of memory.

We also have some auxiliary functions that aid us in the test generation process. The function `adjust_addr` will take an offset and a string representation of a C expression (which evaluates to a pointer) and yield a string representation that adjusts the address to the beginning of the block by subtracting the offset from it. The function `ofs_addr` will take an offset and a string representation of a C expression (which evaluates to a pointer) and yield a string representation that adjusts the address to point to the specific cell in the specified offset by adding

```
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
```

FIGURA 5.2: DFS for test generation

it to the address. Finally, the function `base_var_name` given a natural number n yields a variable we use for testing which will save the address to the beginning of block n. This variable will always have the prefix `__test_harness_x_` plus the number n. For example, for block number 2 the function will yield the string "`__test_harness_x_2`".

Previously we stated that pointers should be followed until we reach either an integer or null value or until we reach a pointer which we already followed. In order to do this we must follow pointers in a certain order and maintain a set of already *discovered* blocks of memory. This way when we find a pointer to a block of memory we already checked (or *discovered*) we can stop and compare the pointer values instead of following the pointers in a cyclic manner indefinitely re-checking parts of the memory which we already checked. We follow the pointers in depth-first search (DFS). In figure 5.2 we have the algorithm used for following a pointer. The algorithm takes a set of natural numbers which are the blocks we have already discovered, an address (the one we are following) and a string representation of the expression in C (which should contain this same address). It yields a new set of discovered blocks and a list of test instructions we have to generate.

The algorithm operates as follows: First, we try to index the block, to see whether the memory is free or holds some content. If the memory is free we will return the same discovered set and will generate no extra instructions. Next, we will adjust the address to the start of the block, this is because when checking the memory we want to check the complete blocks since they are a part of the reachable memory. If the block we are currently checking is already in the discovered set we will simply return the same discovered set and an `Assert_Eq_Ptr` instruction to check the pointers. Howefer, if the block we are currently checking is a new block we have not *seen* yet, we proceed to insert it to the discovered set and add a `Discover` instruction to the list of instructions generated.

We will then proceed to check the contents of the block of memory, starting from the first cell up until the last cell of that block. When checking each cell we will check whether the content of the cell is an integer value, a null value or an address. If the cell contains an integer value, we return the same discovered set and we add an `Assert_Eq` instruction to the list of test instructions to generate. If the cell contains a null value, we return the same discovered set and we add an `Assert_Eq_Null` instruction to the list of test instructions to generate. Finally, if the cell contains an address, we will proceed to follow that address and do a recursive call to `dfs` before continuing to check the current memory block. Upon return of this call, we will return the new discovered set from the recursive call and append the list of test instructions we had generated so far to the list of instructions that the recursive call returned.

5.3. Test Harness in C

In order to support the testing in C we need to have some macros that will correspond to the ones generated in Isabelle. We write a C header file where we define the macros necessary to do the tests as well as some useful variables.

This header file can be seen in figure 5.3. There we can find the definitions for the macros that correspond to the `Discover`, `Assert_Eq`, `Assert_Eq_Null` and `Assert_Eq_Ptr` instructions. For maintaining the discovered set we use a hash set. The implementation of the hash set is done by Sergey Avseyev and it is

```
#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;
```

Figura 5.3: Header file test_harness.h

available online[Avs13]. We also define variables containing the total number of tests, number of passed and failed tests and the discovered hash set.

The test instructions are pretty printed to C macros by using the `test_instructions` function:

```
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"
```

And for each block we generate a `Discover` instruction for, we must also pretty print a declaration for each of the variables we use for testing. This is done as follows:

```
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"
```

Finally, we can get the list of test instructions that must be generated by using `emit_global_tests`. Given a list of variables it will generate a list of test instructions which we will generate C code for. This function is defined as follows:

```
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. Tests

5.4.1. Generation of code with tests

In section 4.9 we described a way of exporting C programs. We have a second way to export C programs where we additionally generate C code for testing the equality of final states.

Previously, we defined the way in which every construct necessary for tests is pretty printed, now we proceed to describe how this test code is generated.

We have a function similar to `prepare_export` that prepares a program for exporting code with tests, it is defined in figure 5.4 (where ⇓ stands for the new line character). First, we obtain the code for the program without tests by using the

```
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 5.4: Function that prepares a program for test export

`prepare_export` function. We only generate test code for valid programs whose execution yields a final state, therefore we must check that by executing the program. Then we create the list of tests for the global variables. Later we generate the string for the variable declarations, a string for initializing the hash set and the string for the actual calls to the macros defined in C. We have three variables in our test harness in C which keep count of how many tests were executed, how many tests were failed and how many tests were passed. We want to get some information about the result of running the tests we generated for the code. In order to do so we generate a piece of code which will print to the standard output (upon execution of the program) the results of testing, i.e. number of tests passed and failed. Finally, `prepare_test_export` will yield a tuple which contains the code for the program without tests, and the respective tests for it.

In order to export the code with the added tests, we define a new ML function called `generate_c_test_code`, its definition is on figure 5.5. The first parameter of the function is a `(string, string) option` which corresponds to the tuple containing the string representation of the program in C code and the tests for that program, respectively. If the function receives a `None` value, this means the `prepare_test_export` function failed and we do not want to generate a C program for it. The second parameter is the path to the directory where we want the program to be exported. This parameter is a relative path to the directory where the theory containing the pretty printing directives is. The third parameter is the name of the program. The last parameter is the theory context we are in.

If the path given is an empty string the generated C code with tests will be pretty printed to Isabelle's output view. The `generate_C_test_code` function will create a new file with the name indicated in the parameters with an extra

```
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 = ()
```

Figura 5.5: Generation of C code with tests

”`.c`”(i.e. < *name* > `.c`) added in the directory indicated by the path parameter. The user will then be able to find the generated code in the directory indicated. This function works by writing the C code for the program in a file and appending the tests we generate for the program at the end of the main function.

The function `expect_failed_test` is very similar to the `expect_failed_export` function presented in section 4.9 but with a different error message. This function will generate an error in Isabelle if code is generated for the program and the tests when we expect the translation process to fail. The function will do nothing when the code is not generated.

5.4.2. Incorrect tests

Considering incorrect cases is important when developing a test suite. We wrote a set of programs for which we expect code generation to fail. By using the functions `expect_failed_export` and `expect_failed_test` defined in sections 4.9

and 5.4.1, respectively, we can write incorrect programs and when generating C code for them we can instruct Isabelle to expect those processes to fail and not to raise an error.

Having incorrect programs is very useful because they serve as regression tests. When adding new features to our semantics we can run all the tests in our test suite. If any of those programs is successfully executed in our semantics and C code is generated, we will detect an error in Isabelle that indicates code is being generated for a program we expect to fail. These tests will be useful for detecting errors if the changes we make change the semantics we had.

However, it is important to note that in order for a regression test suite to be useful it must cover as many cases as possible, which, when working with a bigger language than Chloe, requires a substantial amount of tests written.

5.5. Example programs

In addition to the tests described in this section, we present a set of example programs in Chloe. These are meant to show how programs are written in Chloe and how the execution and code generation work. The list of example programs included in the source code are:

- Bubblesort: implementation of the bubblesort sorting algorithm.
- Count: implementation of a function that counts the occurrences of an element in an array.
- Cyclic linked list: implementation of a cyclic single linked list.
- Factorial: implementation of the factorial function.
- Fibonacci: implementation of a function that computes the Fibonacci number of a given number.
- Linked list: implementation of a single linked list.
- Mergesort: implementation of the mergesort sorting algorithm.
- Minimum: implementation of a function that returns the minimum element of an array.
- Quicksort: implementation of the quicksort sorting algorithm.

- Selectionsort: implementation of the selection sort sorting algorithm.
- String length: implementation of a function that computes the length of a string ending in zero.

5.6. Running Tests

5.6.1. Running tests in Isabelle

In the source code submitted with this work we have a directory which includes all the tests and example programs written for Chloe. We require a way of running those tests in Isabelle automatically.

In the source code we have included an Isabelle theory called "`All_Tests.thy`" which is simply an Isabelle theory that imports every test written for Isabelle, both incorrect and correct. When we open this file in Isabelle all the theory files corresponding to the tests and example programs will be loaded. We will then have two cases for every test.

For regular tests and example programs, code will be generated. For incorrect tests, no code will be generated. In the case where an error occurs, whether it is code being generated for an invalid tests, or code not being generated for a valid tests, we will have an error. It is possible to easily view those in the 'Theories' view of Isabelle's graphical user interface since theory files with an error are marked in red.

5.6.2. Running tests in C

When code is successfully generated we will want to compile and run the tests in an automated manner. We have a Makefile that will compile every test in the test suite as well as a bash shell script which will run every test in the test suite. The result of running the tests will be the number of tests passed and/or failed. When a test fails, the output is printed in red to make it more visible to the user. Additionally, other behaviors are caught by the script and shown to the user, such as out of memory errors, segmentation faults and programs that exited with any of the reserved exit codes[Coo14]. This script is presented in figure 5.6.

```
#!/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
```

Figura 5.6: Shell script for running tests

5.7. Results

Having a test suite that enabled us to check whether our translation process was being done correctly was remarkably valuable during the execution of this work. Originally, we had simple programs written in the language which helped us to intuitively verify that the translation process was done correctly and the semantics of the program was not changed. Subsequently, we were required to automate the testing process and design more specific tests. We proceeded to add the test harness and to create the battery of tests. The results we obtained from the tests were positive. All the incorrect test cases failed as expected. For all the correct cases we generate code with tests and all the tests generated for these programs run successfully.

It is important to note that the test cases were written by us, so we cannot state exact metrics about the results we obtained during the testing process. It is possible that some cases are not completely covered by our test suite since the tests were written by us. This is why new tests can, and should, be added to the test suite in order to keep increasing the trust in the translation process. New tests should also be added as new functionality is added to the semantics.

Capítulo 6

Conclusion and Future Work

6.1. Conclusions

In this work we have managed to successfully formalize the semantics for an imperative language called Chloe which covers a subset of the C language. Chloe has the following features: variables, arrays, pointer arithmetic, while loop construct, if-then-else conditional construct, functions and dynamic memory. We have formalized a small-step semantics in the Isabelle/HOL theorem prover for Chloe and proven the semantics to be deterministic. Additionally, we present an interpreter for the language, thus allowing for programs written in Chloe to be executed within the Isabelle/HOL environment. In order to do so we needed the determinism proofs for the semantics.

Another result of this work was to present a code generator for the Chloe language. This code generator will translate programs from the formal semantics to real C code which can be executed. We guarantee that the generated program can be compiled and executed in a machine given that it is a program without undefined behavior, which complies with the restrictions assumed in the formalization of our semantics (section 3.5) and does not run out of memory during execution. Moreover, we will only generate C code for programs that are valid and execute correctly in our interpreter.

We faced some problems when formalizing the semantics such as allowing only programs with defined behavior to be executed in our semantics. This means we had to detect undefined behavior, according to the C standard[ISO07], such as integer overflow and consider it an error in our semantics. We also faced a problem when formalizing the semantics for the memory allocation function because we assume an unlimited amount of memory, whereas the resources in a machine are limited. In order to solve this problem we decided to change the way a memory allocation call would be translated to C code. We wrap C's malloc function in a new one defined by us that would abort the program if a call to malloc fails. We also had to assume architecture restrictions over the target machine where the generated code will be executed and generate code that verifies if the machine satisfies our assumptions.

Finally, we also present a test harness and a test suite for testing the translation process. The goal of this test suite is to increase the trust in the translation process and to make sure that the semantics of a Chloe program does not change when translated to C code. In this test suite we include incorrect cases that cover all the border cases or cases where no code should be generated due to the program facing an error. Furthermore, we include example programs to demonstrate how the language works and how the programs are translated to C code. Additionally, there is a set of correct tests written that are meant to generate code. The complete battery of tests can be found in the source code submitted with this work.

To guarantee that the semantics of a program is not changed by the translation process, we include a way of generating code with tests. These tests are meant to check that the final state of the program executed in the interpreters within Isabelle/HOL is the same as the state the program is in after being compiled and executed. Two states are equivalent when the global variables and the reachable dynamic memory, at the end of execution, have the same content. For this purpose we wrote a test harness in Isabelle which translates the tests to be made into a set of C macros that we define outside of Isabelle/HOL, and include in our compilation process, which test the equivalence between final states. In order to not run these tests manually we defined a bash script which runs all the existing tests automatically.

6.2. Future work

In this section we will point some directions in which this work can be taken in the future. The time frame available for doing this work made it impossible to include these features in the scope of our work.

First of all, we can upgrade the testing process by parsing C tests from an external test suite and translating them to Chloe. This would allow us to provide more precise metrics for the results of the testing process, e.g. how much of the test suite is successfully covered by our work. An example of this would be to take a test suite made for C compilers such as gcc's C testsuites[StGDC15] and narrow the tests to ones that can be translated to our semantics, translate them from C to Chloe and generate the code with tests from them. After having the translation to Chloe we can generate code and tests for them in order to enhance the current test suite we have provided with this work.

Another interesting direction this work might take is to formalize an axiomatic semantics in order to reason about programs and their properties in Chloe. Since Chloe has pointers as a feature it would be necessary to formalize a separation logic[Rey02] in order to reason about pointers in programs. By extending the work in this direction it would be possible to show partial and total correctness properties proofs of our programs.

One of the features not included in the scope of our work is a proven sound and correct static type system. This would allow to reason about type safety for programs written in Chloe.

There is the Isabelle Refinement Framework[Lam12], which provides a way of formulating non-deterministic algorithms in a monadic style and refine them to obtain an executable algorithm. It provides tools for reasoning about these programs. Another source of future work would be to link our language to the Isabelle Refinement Framework, so that programs from the framework can be refined to programs in Chloe.

Finally, the set of features that are currently supported by Chloe is limited. Another way to improve the results from this work is to expand the set of features supported by Chloe. This might include expanding the set of expressions and instructions (e.g. adding support for structs and unions), adding I/O operations or support for concurrency.

Bibliografía

[Avs13] Sergey Avseyev. Hash set c implementation, 2013. https://github.com/avsej/hashset.c, [Accessed: 2015-08-07].

[BJLM13] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. 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, 2013.

[BL09] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[Coo14] Mendel Cooper. *Advanced Bash-Scripting Guide*, 2014. http://www.tldp.org/LDP/abs/html/index.html.

[dt15] The Coq development team. *The Coq proof assistant reference manual.* TypiCal Project, April 2015.

[GLAK14] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 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, June 2014.

[Haf15] Florian Haftmann. *Code generation from Isabelle/HOL theories*, May 2015. http://isabelle.in.tum.de/dist/Isabelle2015/doc/codegen.pdf.

[ISO07] ISO/IEC. *Programming Languages — C.* ISO/IEC 9899:TC3, 2007.

[Lam12] Peter Lammich. Refinement for monadic programs. *Archive of Formal Proofs*, January 2012. http://afp.sf.net/entries/Refine_Monadic.shtml, Formal proof development.

[LB08] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[Loc13] Andreas Lochbihler. Native word. *Archive of Formal Proofs*, September 2013. `http://afp.sf.net/entries/Native_Word.shtml`, Formal proof development.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014.

[NN07] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer, 2007.

[Nor98] Michael Norrish. *C formalized in HOL*. PhD thesis, University of Cambridge, 1998.

[NPW15] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order-Logic*, May 2015. `http://isabelle.in.tum.de/dist/Isabelle2015/doc/tutorial.pdf`.

[NS14] Michael Norrish and Konrad Slind. *The HOL System Description*, November 2014.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*, July 2002. `http://www.cs.cmu.edu/~jcr/seplogic.pdf`.

[ST14] Christian Sternagel and René Thiemann. Haskell's show class in isabelle/hol. *Archive of Formal Proofs*, July 2014. `http://afp.sf.net/entries/Show.shtml`, Formal proof development.

[StGDC15] Richard M. Stallman and the GCC Developer Community. *GNU Compiler Collection Internals*. Free Software Foundation, Inc., 2015. `https://gcc.gnu.org/onlinedocs/gccint.pdf`.

[Ten91] R. D. Tennent. *Semantics of Programming Languages*. Series in Computer Science. Prentice Hall International, 1991.

[Wen15] Makarius Wenzel. *The Isabelle/Isar Implementation*, May 2015. `http://isabelle.in.tum.de/dist/Isabelle2015/doc/implementation.pdf`.

Apéndice A

@nombreApendice

A.1. @sección

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

A.1.1. @subsección

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

"Saludo".

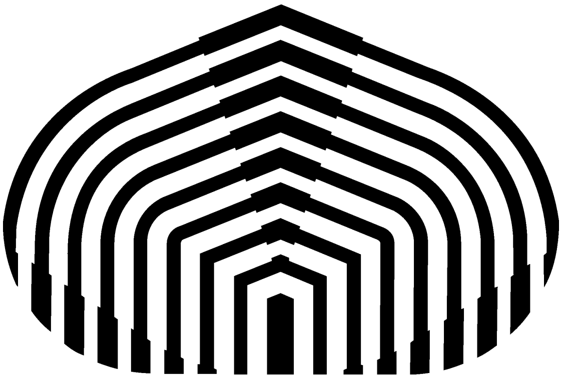

Figura A.1: Grafo gris.

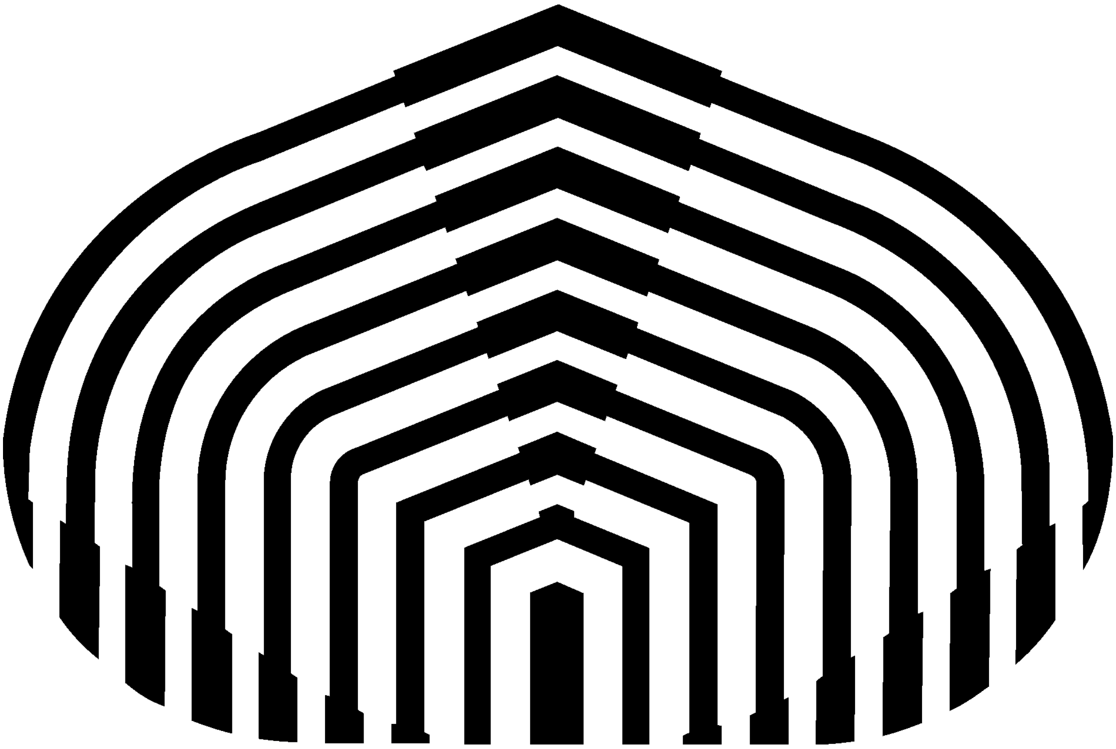

Figura A.2: Grafo con color.

Apéndice B

@nombreApendice

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.