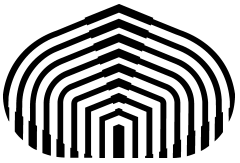


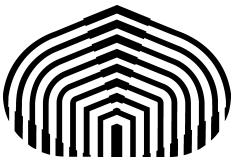
Universidad Simón Bolívar
Decanato de Estudios Profesionales
Coordinación de Ingeniería de la Computación

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

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

Dedicatoria

A @personasImportantes, por @razonesDedicatoria.

Capítulo 1

Introducción

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

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

Finalmente, se describe el contenido del resto del trabajo.

1.1. Motivación

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

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

Desafortunadamente, parte de la semántica para el lenguaje C descrita en el estándar [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 determinística, lo que permite la definición de un interpretador que pueda, efectivamente, ejecutar la semántica. Este interpretador retornará el estado final resultante de la ejecución de la semántica.

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

Finalmente, también es objetivo de este trabajo verificar que la semántica sea compatible con un compilador de C real. Para ello se define un arnés de pruebas y una *suite* de pruebas que tienen el propósito de aumentar la confianza en el proceso de traducción, es decir, la semántica del programa no es cambiada por el proceso de traducción a lenguaje C. El proceso de"*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 para el arnés de pruebas en C que se utiliza para realizar pruebas generadas automáticamente (para los programas escritos en la semántica) que se encargan de comparar el estado final de la semántica ejecutada en el ambiente Isabelle/HOL con aquel del programa compilado.

1.2. Marco Teórico

1.2.1. Semántica de un lenguaje de 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 la secuenciación de instrucciones. Sintácticamente este programa está compuesto por tres instrucciones

separadas por ";", donde cada instrucción está compuesta por una variable, el símbolo ":=" y una segunda variable.

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

Tipos de semántica

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

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

Semántica operacional

Una semántica se define utilizando el enfoque operacional cuando el foco se pone en *cómo* se ejecuta un programa. Se puede considerar como una abstracción de la ejecución del programa en una máquina [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:

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

donde se utilizan las siguientes abreviaciones:

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

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

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

donde se utilizan las siguientes abreviaciones:

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

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

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

Lo que sucede aquí es que en en primer paso, la instrucción $z := x$ se ejecuta y el valor de la variable z cambia a 5, x e y permanecen sin cambios. Luego de la ejecución de la primera instrucción, queda el programa $x := y; y := z$. Se ejecuta la segunda instrucción $x := y$ y el valor de x cambia a 7, y y z permanecen sin cambios. Entonces, queda el programa $y := z$, 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 $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$:

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

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, plegamiento de constantes (*constant folding*) y "accesabilidad" [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.

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

donde se utilizan las siguientes abreviaciones:

$$p_0 = x = n \wedge y = m$$
$$p_1 = y = m \wedge z = n$$
$$p_2 = x = m \wedge z = n$$
$$p_3 = x = m \wedge y = n$$

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

1.2.2. Isabelle/HOL

Hoy en día se cuenta con la existencia de demostradores de teoremas automatizados que ayudan en la formalización y demostración de diferentes programas. Las pruebas escritas en papel y lápiz son propensas a errores y los seres humanos son mas fáciles de engañar que a una máquina. Por lo tanto se deben aprovechar los recursos ofrecidos por una máquina para permitir que realice el trabajo pesado. Razonar sobre la semántica de un lenguaje de programación sin el uso de herramientas 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* 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 de arriba a abajo (*top-down*) donde se tiene la intención de generar código en C de una especificación formal. Existe otra dirección que es relevante mencionar. El proyecto AutoCorres [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 abstracción del *heap* que permite el razonamiento sobre memoria para funciones *type-safe* así como una abstracción para palabras que permite que palabras de la máquina puedan ser abstraídas a números naturales y enteros, de modo que sea posible razonar acerca de los mismos a este nivel.

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

Capítulo 3

Sintaxis y Semántica

El lenguaje imperativo de este trabajo se llama Chloe y representa un subconjunto del lenguaje C. Un programa es una secuencia de una o más instrucciones (o comandos) escritos para llevar a cabo una tarea en una computadora. 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 (*l-values*) y expresioness del lado derecho (*r-values*). En las siguientes secciones al referirse a expresiones del lado derecho o *r-values* se utilizará simplemente el nombre expresiones y se utilizará expresión del LHS (por sus siglas en ingles *left-hand-side*) en lugar de *l-values* o expresiones del lado izquierdo al referirse a dichas expresiones.

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

```
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

son los operadores `*`, `&` y `[]`, respectivamente. Finalmente, como expresiones del LHS se tienen las operaciones de desreferencia 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).

```
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

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

- `new_block` es la función que se encarga de asignar un nuevo bloque de memoria dinámica de un tamaño determinado. Esta función fallará en el caso donde el tamaño dado sea menor o 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 de manera no determinística cuando esta función puede retornar `null`. El problema con esta opción es que complicaría el proceso de demostración de propiedades de un programa ya que cualquier llamada a la función de asignación de memoria dinámica podría fallar. Otra opción sería suponer que se cuenta con una cantidad fija de memoria está disponible. Sin embargo, modelar este tipo de función no es una tarea trivial y permanece fuera del alcance de este trabajo.

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

3.1.2. Semántica

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

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

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

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

Una valuación es una función que "mapea" un nombre de una variable a un valor. El tipo de retorno es `val option option`, lo que modela los siguientes

estados para el valor de una variable: no definida, no inicializada y posee un valor. Por lo tanto, dado un nombre de una variable, esta función puede producir uno de los siguientes resultados:

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

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

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

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

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

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

donde `eval`, dada una expresión y un estado visible, retornará el valor de esta expresión y el estado visible resultante de evaluar dicha expresión. La función `eval_l`, dada una expresión del LHS y un estado visible, retornará el valor de esta expresión (el cual debe ser una dirección) y el estado resultante luego de evaluar dicha expresión. Es importante tener en cuenta que el tipo de retorno de las funciones de evaluación es el tipo `option`. Esto es debido a que estas funciones pueden fallar. Un fallo puede ocurrir en cualquier momento al evaluar una expresión y si

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

FIGURA 3.4: Funciones auxiliares para eval y eval_l

un fallo es encontrado entonces el mismo se propaga hasta que toda la evaluación de la expresión devuelva un valor `None` que indica un error en la evaluación.

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

Las funciones `eval` y `eval_l` dependen de otras funciones definidas con el find e calcular correctamente los valores de las expresiones. La definición de todas las funciones auxiliares para `eval` y `eval_l` se encuentra en la figura 3.4. A excepción de `div_towards_zero` y `mod_towards_zero`, cada una de estas operaciones pueden fallar, por lo que su tipo de retorno es τ `option`. Los valores de este tipo son o `None` cuando la operación falla o `Some`(v) donde v es de tipo τ.

Las funcionalidades de las funciones auxiliares son las siguientes:

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

- La función `read_var` calcula el valor de una variable. Esta función falla cuando el nombre de la variable dado como parámetro corresponde a una variable no definida. Al ser ejecutada exitosamente, la función retorna el valor de la variable. Con el fin de calcular el valor de dicha variable, esta

función chequea la valuación de las variables locales en el estado visible y retorna el valor de la variable si se encuentra definida allí. En el caso en el que la variable no se encuentre definida en el alcance local, la función procederá a chequear el alcance global y retornará el valor de la variable.

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

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

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

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

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

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

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

- La función `mult_val` calcula el valor de la multipliación entre dos valores. La función falla cuando se detecta "overflow" o cuando operandos distintos a dos enteros son dados como parametros a la función. Al ser ejecutada exitosamente, la función retorna un valor entero correspondiente a la multiplicación de los operandos de la función.

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

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

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

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

```
type_synonym fname = string

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

FIGURA 3.5: Comandos en Chloe

parámetros, la función retorna un valor entero `I` 1 si ambas direcciones son iguales y un valor entero `I` 0 de lo contrario. Dos direcciones son consideradas como iguales cuando ambos componentes de la tupla son iguales.

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

3.2. Comandos

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

3.2.1. Sintaxis

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

Aquí se procede a describir la *sintaxis abstracta* de loc omandos en el lenguaje Chloe.

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

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

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

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

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

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

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: Equivalencia entre sintaxis abstracta y concreta

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

valid_fun_decl :: fun_decl ⇒ bool
```

FIGURA 3.6: Definiciones de funciones

3.3. Funciones

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

Como es visible en la definición de la figura 3.6, una función consiste en un nombre, los parámetros formales, las variables locales y el cuerpo de la función, que es un comando, potencialmente grande, en el lenguaje Chloe. También se define un predicado que comprueba si una declaración de una función es válida o no. La declaración de una función se considera válida si y solo si los parámetros de la función y las variables locales tienen nombres distintos.

3.4. Programas

Un programa en Chloe consiste en un nombre, una lista de variables globales y una lista de funciones como se muestra en la figura 3.7. En esa misma figura se encuentra la definición de una condición que todo programa válido debe cumplir.

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

- Los nombres de las variables globales son distintos entre si.
- Los nombres de las funciones en el programa son diferentes entre si.
- Cada declaración de función para cada función en el programa debe ser válida.
- La función main debe estar definida.

```
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: Definiciones de un programa

- Ninguno de los nombres de las variables o nombres de funcion en el programa debe ser una palabra clave reservada del lenguaje C o una palabra clave reservada para *testing*.[2]

- Las variables globales y los nombres de las funciones en un programa no pueden ser iguales.

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

3.5. Restricciones

Esta semántica hace una suposición con respecto a la máquina en la que el código va a ser ejecutado. Esta restricción está parametrizada y corresponde a la arquitectura de la máquina en la que el código será ejecutado. Como se dijo anteriormente, la precisión de los valores enteros puede ser modificada con el fin de hacer que esta semántica sea compatible con diferentes arquitecturas, por ejemplo arquitecturas de 32 bits.

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

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

Otra restricción en la semántica es que solo funciona para un subconjunto de C donde la semántica es determinística y cualquier comportamiento indefinido se considera como un error en la semántica. Se pasará a un estado erróneo si la semántica encuentra comportamiento que sea considerado como indefinido por el documento de referencia del lenguaje C [ISO07], un ejemplo de este tipo de comportamiento es "overflow" de enteros.

3.6. Estados

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

3.6.1. Valuación

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

3.6.2. Pila de ejecución

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

En la figura 3.8 el tipo definido para un marco de pila es una tupla que contiene un comando en Chloe, una valuación y una ubicación de retorno. El comando de Chloe corresponde al cuerpo de la función a ejecutar. La valuación corresponde a las variables locales a la función que fue llamada. Finalmente, la ubicación de retorno puede ser una de las siguientes: una dirección, una variable o una ubicación de retorno inválida. Cuando una función retorna una variable pueden suceder varias cosas:

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

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

La ubicación de retorno se encuentra en el marco de pila del llamador, es decir, al retornar de una función el marco del llamador es el que debe ser revisado para saber a donde asignar el valor de retorno o si un valor de retorno es esperado. Para aclarar esto podemos tomar el ejemplo de código en la figura **??**. Es un programa sencillo donde una función que suma el valor de sus dos parámetros se define y luego se llama desde la función main. Antes de la llamada a función, el marco de pila perteneciente a la función main tiene una ubicación de retorno *Invalid* y luego

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

type_synonym stack_frame = com × valuation × return_loc
```

FIGURA 3.8: Stack definitions

de la llamada a función la ubicación de retorno cambia a la variable x. Observe que el marco de pila que cambia su ubicación de retorno es aquel correpondiente a main, esto es porque el llamador es el que debe guardar la ubicación de retorno donde espera guardar el valor de retorno de una función llamada. Una ubicación de retorno *Invalid* indica que el llamador no espera guardar resultado alguno, es decir, el llamador no ha llamado aun a alguna función o la función llamada no tiene valor de retorno.

3.6.3. Tabla de procedimientos

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

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

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

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

3.6.4. Estado

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

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

3.6.5. Estado inicial

Con el fin de construir un estado inicial se deben definir ciertos componentes. En la figura 3.7 se encuentran definiciones para la configuración inicial de la pila, las variables globales y la memoria. La configuración inicial para la pila consiste en la pila que contiene únicamente el marco de pila para la función main del

```
context fixes program :: program begin

  private definition proc_table ≡ proc_table_of program

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

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

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

end
```

FIGURA 3.9: Construcción del estado inicial

programa. La configuración inicial para la valuación global es una función donde cada posible nombre de variable se mapea al valor correspondiente a una variable no definida (`None`). La configuración inicial de la memoria dinámica es la memoria vacia, dado que nada ha sido asignado.

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

3.6.6. Estado visible

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

El marco de pila en el tope de la pila corresponde a la función actual que está siendo ejecutada y el comando en ese marco de pila corresponde al comando (o programa) en Chloe que está siendo ejecutado. Cada vez que se ejecuta un comando o se toma un paso en la semántica de pasos cortos, este comando se actualiza para

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

contener el próximo comando a ejecutar. Con el fin de demostrar que la semántica de pasos cortos es determinística se debe demostrar que cada vez que se tiene una pila de ejecución no vacía, el orden en que se aplica el transformador de evaluación[4] y la función que actualiza el comando a ejecutar a continuación es irrelevante para el estado final resultante. Es por ello que se introduce una definición separada para un estado visible aparte de la de un estado regular, es una vista del mismo estado completo pero con informacion limitada.

3.7. Semántica de pasos cortos

3.7.1. CFG

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

Enabled functions Una función "enabled" es una función parcial que se define como sigue:

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

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

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

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

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

definition en_pos :: exp ⇒ enabled

definition en_neg :: exp ⇒ enabled
```

FIGURA 3.10: Funciones "enabled"

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

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

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

Transformer functions Una función "transformer" es una función parcial que se define como sigue:

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

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

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

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

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

El "transformador" para una asignación a una celda en memoria producida por `tr_assignl`, se encarga de evaluar la expresión del LHS para obtener la ubicación de memoria donde se guardará el valor, evaluar la expresión para obtener el nuevo valor y guardar el valor en memoria. Retorna el estado resultatnte de realizar dichas evaluaciones y la opración `store`.

El "transformador" para una evaluación producido por `tr_eval`, se encarga de evaluar una expresión. Retorna el estado resultante de realizar dicha evaluación.

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

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

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

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

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

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

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

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

Se tiene una función `tr_return` que produce un "transformador" para una llamada a `return` que retorna una expresión de una funcion, este "transformador"

```
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

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

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

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

```
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

desempilará de la pila cuando exista una instrucción `return`. Este concepto será importante al hablar de llamadas a funciones y retornos de funciones más adelante.

En la figura 3.12 se encuentra una definición inductiva para CFG, donde se pueden ver las reglas para formar aristas entre comandos. Se puede seguir una arista en el CFG desde una asignación a una variable a `SKIP`. Esta arista siempre está "enabled" y el "transformer" para actualizar el estado es el resultado de llamar a `tr_assign` con los parámetros específicos para el comando.

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

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

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

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

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

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

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

3.7.2. Reglas de la semántica de pasos cortos

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

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

Las reglas para la semántica de pasos cortos se encuentran detalladas en la figura 3.13. La notación de infijo para la semántica de pasos cortos se escribe: $s \to s_2$ lo cual significa que se toma un paso corto desde s hasta s_2. Se puede dar un paso cuando las siguientes condiciones se cumplen:

- La pila no está vacía
- Hay una arista en el CFG entre c_1 y c_2.
- El comando en el marco de pila en el tope de la pila en el estado inicial es c_1.
- Al aplicar la función "enabled" sobre el estado, esta retorna `True`.
- Al aplicar el "transformer" al estado con el comando en marco de pila en el top de la pila actualizdo de c_1 a c_2, este retorna un nuevo estado s_2.

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

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

```
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

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

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

- La función "enabled" o el "transformer" retornan `None` al ser evaluados sobre el estado inicial. Esto indica que un estado erróneo fue alcanzado durante la evaluación de alguna de esas dos funciones. El estado erróneo se propaga al tomar un paso corto desde el estado s hasta `None` y luego la ejecución se queda atascada allí.

- Si al aplicar el "transformer" `tr_return_void` sobre el estado el mismo retorna un valor `None`, el comando en el tope de la pila en el estado inicial es SKIP y la pila no está vacía, entonces también se propaga este estado erróneo `None` al tomar un paso corto desde s hasta `None`. Esto indica que hubo un error al retornar de una función sin valor de retorno.

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

También se define (en la figura 3.13) una nueva definición `small_step'` basada en la definición previa para `small_step`. Esta definición basicamente dice que si un paso corto puede ser tomado desde el estado s hasta s' entonces un paso corto

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

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

Determinism

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

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

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

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

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

En el caso de `If` se debe hacer una prueba por casos sobre el valor de `en_pos` $b\ s$. Puede fallar y retornar un valor `None`, entonces el caso está resuelto. Si no falla entonces se debe chequear el valor booleano retornado por la función `en_pos`. En

el caso donde es `True` entonces el caso está resuelto. El caso difícil es cuando la evaluación de `en_pos` es `False`, semanticamente, esto significa que se debe tomar la rama del else. Por lo tanto cuando `en_pos` b s es `False`, `en_neg` b s siempre se evaluará como `True`, esto significa qe el comando siempre tendrá una arista habilitada que puede seguir, es decir, `en_neg` b s y la demostración de este caso está completa. □

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

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

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

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

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

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

En el siguiente caso se supone `cfg` c (en, tr) c' y `en` $s\ =$ `SomeTrue`. Este es el caso donde hay una arista que está habilitada para ser seguida y aún se deben

revisar dos casos mas. Aplicar el "transformer" sobre el estado actualizado con el comando (tr (upd_com c' s)) puede fallar o no. En el caso donde falla (retorna `None`) se puede tomar un paso a `None` y se habrá probado que hay un estado al que s puede tomar un paso corto, es decir `None`. En el caso donde no falla, retornará `Some` s_2. En este caso se puede tomar un paso a `Some` s_2 y se habrá probado que hay un estado al cual s puede tomar un paso corto, es decir `Some` s_2. □

Se definen varios lemas que serán útiles más adelante.

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

Lema 3.3 (cfg_SKIP_stuck)**.**
$\neg$ `cfg SKIP` a c

Demostración. La propiedad es demostrada automáticamente. □

Lema 3.4 (ss_empty_stack_stuck)**.**
`is_empty_stack` $s \Longrightarrow \neg\ (s\ \rightarrow\ cs')$

Demostración. La propiedad es demostrada automáticamente. □

Lema 3.5 (ss'_SKIP_stuck)**.**
`is_empty_stack` $s \Longrightarrow \neg\ (Somes\ \rightarrow\ cs')$

Demostración. La propiedad es demostrada automáticamente. □

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

Lema 3.6 (en_neg_by_pos)**.**
`en_neg` e s = `map_option (HOL.Not) (en_pos` e s `)`

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

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

Lema 3.7 (cfg_determ)**.**
`cfg` $c\ a1\ c' \wedge$ `cfg` $c\ a2\ c''$
$\Longrightarrow a1\ =\ a2 \wedge c'\ =\ c'' \vee$
$\exists\ b.\ a1\ =$ (`en_pos` b, `tr_eval` b) $\wedge\ a2\ =$ (`en_neg` b, `tr_eval` b) $\vee$
$\exists\ b.\ a1\ =$ (`en_neg` b, `tr_eval` b) $\wedge\ a2\ =$ (`en_pos` b, `tr_eval` b)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.8. Interpretador

3.8.1. Ejecución de un solo paso

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

Un paso *Base* tiene un "transformer" y siempre está habilitada para seguir hasta un nuevo comando y un paso *Cond* que, además del "transformer", también tiene una función "enabled" y dos comandos. Esta función "enabled" indica si se debe tomar un paso al primer o al segundo comando. También se define una función `cfg_step` que dada el comando de inicio retorna que tipo de paso sigue en el CFG.

La función `fstep`, definida en la figura 3.15, indica como se toma un solo paso el la ejecución de la semántica. La ejecución de un paso en la semántica va desde un estado inicial a un nuevo estado que puede ser erróneo (`None`) o uno válido (`Some` s). Para ejecutar un comando `SKIP` se llama a `tr_return_void`. En cualquier

```
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: Aristas de un solo paso

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

FIGURA 3.15: Definición de fstep

otro comando se verifica que tipo de paso se debe tomar utilizando la función `cfg_step` y, basado en esto, se decide que hacer. Si es un paso `Base` se llama al "transformer" sobre el estado con el comando actualizado. Si es un paso `Cond` se evalúa la condición y se llama a la función transformer sobre el estado con el comando actualizado.

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

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

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

Lema 3.12 (fstep1)**.**
$s \to s' \implies$ `fstep` $s = s'$

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

Luego se considera la dirección opuesta:

Lema 3.13 (fstep2)**.**
$\neg$ `is_empty_stack` $s \implies s \to$ (`fstep` s)

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

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

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

3.8.2. Ejecución e interpretación

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

La definición para tal interpretador se encuentra en la figura 3.16. Primero se define el criterio para considerar un estado como final. Un estado será considerado como final cuando su pila de ejecución esté vacia o cuando sea `None`.

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

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

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

FIGURA 3.16: Definición de un interpretador para Chloe

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

FIGURA 3.17: Definición de un interpretador para Chloe

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

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

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

Para poder demostrar esto necesitamos un lema que "unfolds the loop" en la definición de nuestro interpretador:

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

Demostración. La demostración es hecha automáticamente. □

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

Solamente programas válidos pueden ser ejecutados. En la figura 3.17 se encuentra la definición de la función que ejecuta un programa. Para ejecutar un programa se debe asegurar que el mismo es válido utilizando el criterio definido en `valid_program` y luego se interpreta el estado inicial del programa p.

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

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

FIGURA 3.18: Definiciones sobre ejecución de programas

3.8.3. Correctitud

Por último, se debe demostrar que el interpretador es correcto. En la figura 3.18 se encuentran dos definiciones con respecto a la ejecución. La primera establece que una ejecución de un estado *cs* produce (`yields`) *cs′* si se pueden tomar pasos cortos desde *cs* hasta *cs′* y *cs′* es un estado final. En segundo lugar, se dice que la ejecución de un estado *termina* si existe algún estado *cs′* tal que sea producido por la ejecución del estado *cs*.

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

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

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

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

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

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

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:

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

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) (`Const`5)	(2 + 5)
`Subst` (`Const` 9 (`Const`5)	(9 − 5)
`Minus` (`Const` 9)	(−9)
`Div` (`Const` 8) (`Const` 4)	(8 / 4)
`Mod` (`Const` 8) (`Const` 4)	(8 % 4)
`Mult` (`Const` 9) (`Const` 3)	(9 ∗ 3)
`Less` (`Const` 7) (`Const` 9)	(7 < 9)
`Not` (`Const` 0)	! (0)
`And` (`Const` 1) (`Const` 1)	(1 && 1)
`Or` (`Const` 1) (`Const` 0)	(1 ‖0)
`Eq` (`Const` 6) (`Const` 4)	(6 == 4)
`New` (`Const` 9)	(`intptr_t`) __`MALLOC`(`sizeof`(`intptr_t`) ∗ (9))
`Deref` (`V` foo)	∗((`intptr_t` ∗) foo)
`Ref` (`V` foo)	((`intptr_t` ∗) &(foo))
`Index` (`V` bar) (`Const` 3)	((`intptr_t` ∗) bar[3])
`Derefl` (`V` foo)	(∗((`intptr_t` ∗) foo))
`Indexl` (`V` bar) (`Const` 3)	((`intptr_t` ∗) bar[3])

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	$*((\texttt{intptr\_t}*)\ foo) = 4;$
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 $(\&\ (*((\texttt{intptr\_t}*)\ foo))$;
RETURN (V foo)	return $(foo))$;
RETURNV	return;
(Derefl foo) ::== bar ([V baz, Const 4])	$*((\texttt{intptr\_t}*)\ foo) = bar(baz,\ 4);$
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 $< arg\_name_0 >$, ..., intptr_t $< arg\_name_n >$) {

  intptr_t $< local\_var_0 >$

⋮

  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

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

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".

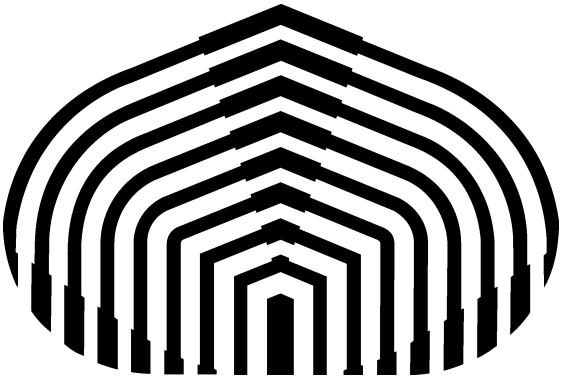


FIGURA A.1: Grafo gris.

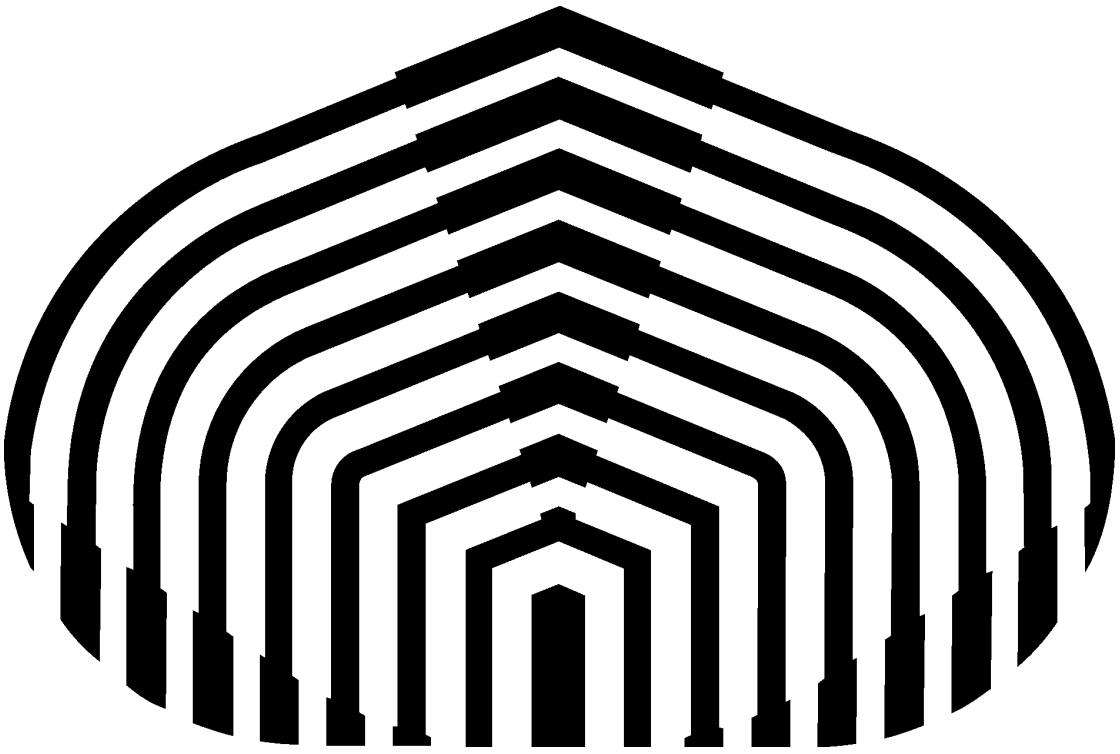


FIGURA A.2: Grafo con color.

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.