ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Trabajo fin de Grado

Ingeniería del software

Teoría de Lenguajes de Programación y Compiladores

Realizado por
Rodrigo García Casasola

Dirigido por
José Ra. Portillo Fernández

Departamento
Matemática Aplicada I

Sevilla, 20 de Junio de 2022

Resumen

En este trabajo de fin de grado se construye un compilador para el lenguaje de programación imperativa IMP. Utilizando un subconjunto de instrucciones de la JVM *(Java Virtual Machine)*. Para después validar que la semántica se mantiene mediante la verificación formal de propiedades lógicas del compilador.

Posteriormente se le añaden mejoras de optimización a nivel de recursos como un módulo de Deadcode. Que utiliza análisis estático *(static analysis)* para estudiar la viabilidad de las variables del programa *(Liveness analysis)*. También se le añade otro módulo de optimización Regalloc. En este caso de la memoria usada por el programa.

Los objetivos cubren el conocimiento de un compilador y sus tecnologías mediante la programación funcional, la verificación software y la teoría de lenguajes de programación. Todo ello se desarrollará en el lenguaje Coq.

Agradecimientos

A mi guía, mi referente y mi amigo. Tanto en lo personal como en lo profesional. Por ti y por la hermandad que nos une. Por estar a la altura de todo lo que me das y de todo lo que está por venir. Te pienso como a uno mismo.

Índice general

Índice general **III**

1 Introducción **1**

2 Objetivos **2**
2.1 Definición de objetivos . . . 2
2.2 Justificación de los objetivos . . . 3

3 Antecedentes **4**
3.1 Programación Imperativa . . . 4
3.1.1 Definición . . . 4
3.1.2 Contexto histórico . . . 5
3.2 Programación Funcional . . . 6
3.2.1 Definición . . . 6
3.2.2 Contexto histórico . . . 6
3.3 Teoría de Coq . . . 8
3.3.1 Introducción . . . 8
3.3.2 Tipos inductivos . . . 9
3.3.3 Funciones . . . 9
3.3.4 Tipos . . . 9
3.3.5 Tipos recursivos . . . 10
3.3.6 Listas . . . 11
3.3.7 Polimorfismo . . . 11
3.3.8 Funciones de alto nivel (higher-order functions) . . . 13
3.3.9 Mapas . . . 14
3.3.10 Tipo Option . . . 16
3.3.11 Tácticas . . . 17
3.4 Teoría de lenguajes de programación . . . 17
3.4.1 Sintaxis . . . 17
3.4.2 Árboles sintácticos . . . 18
3.4.3 Semántica . . . 18
3.4.4 Operaciones semánticas . . . 18
3.4.5 Demostración por reducción de términos . . . 19
3.5 IMP . . . 20
3.5.1 Introducción . . . 20
3.5.2 Sintaxis . . . 20
3.5.3 Evaluación como función . . . 23
3.5.4 Evaluación como relación . . . 23
3.5.5 Equivalencia de las definiciones . . . 25
3.5.6 Estados y variables . . . 27

3.5.7 Comandos 30

4 Aportación realizada **34**
4.1 Introducción 34
4.2 Semanticas 36
4.2.1 Big-Steps 37
4.2.2 Small-Steps 37
4.2.3 Semánticas big-steps coinductivas para expresiones divergentes 41
4.3 Compilador 46
4.3.1 Instrucciones 46
4.3.2 Transiciones 47
4.3.3 Esquema de compilación 48
4.3.4 Expresiones aritméticas 48
4.3.5 Expresiones booleanas 48
4.3.6 Comandos 49
4.4 Deadcode 49
4.4.1 Conjuntos de variables 49
4.4.2 *Liveness analysis* 49
4.4.3 Eliminación del código muerto 50
4.5 Regalloc 51
4.5.1 Reasignación de las variables 51
4.5.2 Transformación de los comandos 51
4.5.3 Condiciones para la reasignación de variables 51
4.5.4 Función de reasignado de variables 53

5 Análisis de requisitos, diseño e implementación **55**
5.1 Compilador 55
5.1.1 Instrucciones 56
5.1.2 Transiciones 58
5.1.3 Evaluación aritmética 60
5.1.4 Evaluación booleana 61
5.1.5 Evaluación de comandos 62
5.2 Módulo de optimización: Deadcode 63
5.2.1 Comparador de variables 64
5.2.2 Evaluación de las variables 65
5.2.3 Analizador de vivacidad de las variables 67
5.2.4 Analizador de código muerto 68
5.3 Módulo de optimización: Regalloc 69
5.3.1 Evaluación de expresiones 69
5.3.2 Transformación de los comandos 70

6 Análisis temporal y costes **72**
6.1 Análisis temporal del proyecto 72
6.2 Estimación real: análisis temporal y costes 73
6.2.1 Caso general 73
6.2.2 Caso simple 74
6.2.3 Caso de estudio 74

7 Comparación con otras alternativas **76**
7.1 Intérprete 76
7.2 Código máquina 77
7.3 Máquina virtual 77
7.3.1 Implementación de la JVM 77

8 Pruebas **78**
8.1 Preservación semántica: Compil, Big-step coinductivos 78
8.1.1 Lemas necesarios 78
8.1.2 Verificación para expresiones aritméticas 79
8.1.3 Verificación para expresiones booleanas 80
8.1.4 Verificación para comandos: caso terminativo 82
8.1.5 Verificación para comandos: caso divergente 84

9 Conclusiones y desarrollos futuros **87**
9.1 Conclusiones 87
9.2 Propuestas de mejora 88
9.3 Futuros desarrollos: 88

Bibliografía **89**

CAPÍTULO 1

Introducción

Esta introducción sirve para recoger un pequeño resumen de los contenidos a tratar en cada capítulo. De manera que se pueda tener una vista general de lo expuesto y sirva como expansión del índice anterior a nivel de contenido.

En el capítulo de Antecedentes se hará copia de todas las herramientas y conceptos teóricos fundamentales para la compresión y construcción del lenguaje IMP. Que servirá de base para construir el compilador. En cuanto al capítulo Aportación recogerá la licitación de los elementos necesarios para la construir de las reglas semánticas, el compilador, y sus estructuras de optimización: el módulo DeadCode y el módulo Regalloc.

En el capítulo de Análisis de requisitos, diseño e implementación se concretizará los explicitado en Aportación a realizar, materializando para cada componente del sistema los requisitos que deberá cumplir, las decisiones de diseños que satisfacen dichos requisitos y la implementación del sistema mediante fragmento de código.

En el capítulo de Análisis de temporal y de costes se recogen dos enfoques: Primero, se recoge el tiempo estimado y real para la realización de dicho proyecto. Y segundo, se recoge una estimación real a nivel temporal y de costes acerca de la creación real de un compilador. Con el objetivo de ilustrar la envergadura y dimensión real de la creación de un compilador.

En el capítulo de Pruebas se recogen las demostraciones que corroboran que la traducción realizada por el compilador construido preservan la semántica de IMP. Se recogen también lemmas fundamentales necesarios para dichas demostraciones. Estas se realizan con el demostrador de teoremas semiautomático de Coq.

Por último, en el capítulo de Conclusiones y desarrollos futuros se recoge la experiencia y conocimientos adquiridos durante la realización de este proyecto. Así como una lista de posibles mejoras, aportaciones y continuaciones sobre lo ya establecido.

CAPÍTULO 2

Objetivos

2.1– Definición de objetivos

Los objetivos a alcanzar en este proyecto serán:

- Aplicar los fundamentos de la teoría de los lenguajes de programación.
- Analizar la semántica y sintaxis de un lenguaje de programación.
- Analizar la arquitectura básica de un lenguaje de programación desde cero.
- Analizar la creación de un compilador desde cero.
- Aplicar los beneficios de la programación funcional y la verificación formal para sistemas críticos.
- Exponer un caso práctico; la construcción de un compilador, basado en tecnologías actuales como la JVM.
- Trabajar con herramientas de optimización utilizadas en los compiladores actuales.
- Estudiar el lenguaje de demostración semiautomático Coq.

2.2– Justificación de los objetivos

Para poder comprender y analizar el funcionamiento interno de un compilador es necesario conocer previamente los principios de la base sobre la que trabaja; los lenguajes de programación. Por lo tanto partimos en asentar la teoría de lenguajes de programación: Como se crean las expresiones, la semántica y sintaxis, la aritmética, el árbol de sintaxis. Una vez dispuesta, se pondrá en práctica con la creación de un sencillo lenguaje de programación imperativa *(IMP)* sobre el cual creamos un compilador a la *JVM (Java Virtual Machine)* para demostrar que la semántica se mantiene a pesar de que la sintaxis es traducida de lenguaje.

El compilador, al ser un componente crítico en el software, necesita de los beneficios de la programación funcional y la verificación formal para blindar su funcionamiento con la fuerza de la lógica. Por lo tanto también será necesario conocer y aprender las características más interesantes. Para ello se utilizará el asistente de demostración interactivo *Coq*. por lo que será necesario conocer sus estructuras básicas, su funcionamiento y sus bases en una breve introducción citando tan solo lo estrictamente necesario para el caso práctico. También será el medio por el que se define todas las implementaciones a realizar en este proyecto, por lo que su uso y conocimiento es vital.

Una vez conocidas todas estas bases se procederá a optimizar el caso práctico con nuevas herramientas y tácticas para asegurar que su funcionamiento se equipara a la realidad actual en cuanto a rendimiento y estabilidad. Se expondrá la idea intuitiva de dichas herramientas, se definirán utilizando las bases establecidas y se implementarán en el caso práctico utilizand *Coq* para ello.

Con estos objetivos se busca cubrir información acerca de elementos que se utilizan de forma rutinaria en el mundo software y que son la piedra angular de su desarrollo. Pero de los cuales se conoce muy poco debido a que quedan completamente dados por supuestos y automatizados. El compilador es la última frontera con respecto al software y la herramienta que se encarga de traducir el código de alto nivel a bajo nivel. Los lenguajes de programación son el medio de comunicación principal de los desarrolladores software con la máquina. Por lo que estamos hablando de los esenciales a la hora de construir software.

CAPÍTULO 3

Antecedentes

3.1– Programación Imperativa

3.1.1. Definición

Para empezar, se comenzará definiendo el concepto de programación imperativa.

La programación imperativa es un tipo de paradigma de programación en el que se especifican los pasos que el computador debe seguir para completar o terminar un problema a resolver. Está construido con instrucciones o comandos que se explicitan en un orden concreto, cambiando el estado del programa. Entre sus características, destaca:

- fácil lectura y aprendizaje.
- Usa instrucciones para cambiar el estado del programa.
- Las instrucciones siguen un orden secuencial para resolver el problema.
- El orden de ejecución de estas instrucciones es fijo.
- Es menos expresivo y seguro que la programación funcional
- Las estructuras de datos a utilizar necesitan ser representadas como cambios de estados.

Ejemplos de lenguajes de programación imperativa son: Java, C++, Rust, Fortran, etc.

Una vez definido el concepto, se expondrá el contexto histórico del mismo para tener una visión de conjunto y entender su evolución a través del tiempo y de las decisiones tomadas. Se ampliarán los conceptos más relevantes y otros tan solo quedaran explicitados dejando al lector la posibilidad de ampliarlos por su cuenta propia.

3.1.2. Contexto histórico

Los lenguajes de programación imperativa nacen del concepto de la máquina de Turing, definida el matemático e informático teórico Alan Turing en el año 1937.

Esta máquina, definida informalmente, es una máquina que opera mecánicamente sobre una cinta en la que existen símbolos que pueden ser leídos o escritos por la máquina. La operación a realizar está completamente definida por un conjunto finito de instrucciones elementales expresadas como condiciones lógicas. El concepto más importante introducido por esta máquina es el concepto de estado, ya que según su definición, la maquina de turing posee un registro de estado que almacena uno de los estados finitos.

Siguiendo este concepto de estado aparece FORTRAN, desarrollado por John Backus de IBM en 1954, fue el primer lenguaje de programación en solucionar los problemas de ineficiencia del código máquina a la hora de crear programas complejos, FORTRAN era un lenguaje compilado que introdujo variables definidas, expresiones complejas, subprogramas y muchas de las características propias de los lenguajes de programación imperativa.

En la década de 1960, ALGOL fue desarrollado para poder expresar algoritmos matemáticos de manera mas fácil. Introdujo conceptos como: tipo, declaración, identificadores, la sentencia *for*, *while*, *if and else*, bloques de entorno, alcance de la declaración de variables, *arrays* dinámicos, variables locales y globales, etc. ALGOL bebía de FORTRAN y sus contemporáneos: COBOL y BASIC, que también fueron desarrollados en dicha década.

En los 70 Pascal fue desarrollado por Niklaus Wirth, y C fue creado por Dennis Ritchie mientras trabajaba en los laboratorios Bell. El logro más importante de C (accidente histórico) ha sido la tolerancia de los compiladores de C a errores en el tipado. C evolucionó de lenguajes sin tipado, no apareció de repente como un lenguaje enteramente nuevo con sus propias reglas, si no que los programas se tuvieron que ir adaptando de manera progresiva al desarrollo del lenguaje.

En el 1980 aumentó el interés por los lenguajes de programación orientados a objetos, un subtipo de los lenguajes de programación imperativos. Su principal característica es que añadían el concepto de objeto; Una combinación de variables, funciones y estructuras de datos que definen una clase. En las dos últimas décadas del siglo 20 se desarrollaron numerosos lenguajes de programación siguiendo este paradigma. El más destacable, desarrollado por Bjarne Stroustrup en 1983, fue C++, basado en C.

Siguiendo la línea de lenguajes de programación basado en objetos se desarrollaron, Perl, por Larry Wall en 1987. Python, por Guido van Rossum en 1990 y Ruby, por Yukihiro Matsumoto en 1995, por destacar algunos de los muchos creados.

Actualmente es el paradigma de programación más extendido a nivel comercial, ya que sus lenguajes se utilizan ampliamente en la construcción de sistemas software de cualquier índole.

3.2– Programación Funcional

3.2.1. Definición

La programación funcional es un tipo de paradigma de programación creado para resolver problemas de un modo puramente funcional. Es una aproximación totalmente matemática a dichos problemas los cuales se buscan resolver como operaciones e interacciones de funciones, sin cambiar el estado de la máquina ni sus valores. Es una forma de programación declarativa. De sus características destaca:

- Garantiza la no existencia de *bugs*, mejora el rendimiento e incrementa la reutilización y testeo del código
- Usa funciones como unidad elemental de resolución de problemas.
- Los programas normalmente están estructurados como funciones sucesivas y anidadas.
- El orden de ejecución de las instrucciones no es fijo.
- Es más expresivo y seguro que la programación imperativa.
- Requiere la explicitación de las estructuras de datos que se van a utilizar

Ejemplos de lenguajes de programación funcional son: Haskell, Lisp, ML, etc.

3.2.2. Contexto histórico

Los lenguajes de programación funcional nacen del concepto de *lambda calculus* definido por el matemático y lógico Alonzo Church en 1932.

El ***lambda calculus*** es una sistema formal de computación basado en la aplicación de funciones. En 1937 Alan Turing probó que el *lambda calculus* y las máquinas de turing era modelos equivalentes de computación (**Tesis Church-Turing**). el cálculo lambda es el concepto básico de los lenguajes de programación funcionales. Church desarrolló posteriormente una versión más débil del cálculo lambda; el cálculo lambda de tipo simple, que extendía al cálculo lambda introduciendo la asignación de tipos. Esto más tarde sería la base de la programación funcional con tipado estático.

El primer lenguaje funcional de alto nivel fue LISP, desarrollado en los 50 para los computadores científicos de IBM por John McCarthy en el *MIT*. LISP fue definido usando la notación lambda de Church, extendida con un tipo para permitir funciones recursivas. LISP introdujo numerosos conceptos propios de la programación funcional. A pesar de que LISP en un principio era multiparadigma e introdujera soporte a numerosos estilos de programación, lenguajes posteriores, como Scheme, Clokure o Julia, buscaron simplificar y racionalizar LISP mediante un núcleo funcional más sencillo.

En 1956 se desarrolló IPL, el cual era un lenguaje de programación de estilo ensamblador para manipular listas de símbolos. Tenía una noción de generador que equivale a una función que acepta funciones como argumentos y, como era un lenguaje a nivel de ensamblador el código podía tratarse como datos, por lo tanto IPL puede ser visto como un lenguaje con funciones de primer nivel.

Kenneth E. Iversion desarrolló APL a principios de los 60. que fue la influencia principal de FP, presentado por John Backus en 1977. En él, John define que los programas funcionales se construyen en un orden jerárquico mediante çombinaciones de forma"que permitan un "álebgra de programas". En lenguajes modernos, esto significa que los programas funcionales sigan el principio de composicionalidad. Este principio enuncia que el significado de las expresiones complejas está definido por el significado de las subsecuentes expresiones y regladas usadas para crearla.

En 1973 ML fue creado por Robin Milner en la Universidad de Edinburgo. También en Edinburgo se desarrolló NPL. Este último estaba basado en los teoremas de recursión de Kleene. Brustall, MacQUeen y Sannella más tarde incorporaron el tipo polimórfico de ML para producir el lenguaje Hope. ML más tarde acabó derivando en numerosos dialectos, los más conocidos actualmente OCaml y *Standard ML*.

En 1980, Per Martin-Löf desarrolló la teoría de tipado intuitiva (también llamada teoría de tipado constructiva) que asoció la programación funcional con pruebas constructivas expresadas como tipos dependientes. Esto permitió nuevas revisiones de la demostración de teoremas interactiva y ha influenciado en el desarrollo de lenguajes de programación funcional dedicados a ello.

El lenguaje de programación funcional perezosa Mirada desarrollado por David Turner en 1985 tuvo una enorme influencia en el conocido Haskel el cual empezó como un consenso en 1987 para formar un estándar abierto para la investigación de la programación funcional. Su implementación llevan en desarrollo desde 1990.

De manera más reciente la programación funcional ha encontrado un nicho en el *framework* geométrico CSG. En el ámbito comercial, la programación funcional destaca por su uso en sistemas críticos.

3.3– Teoría de Coq

3.3.1. Introducción

Coq es un sistema de ayuda para la demostración de teoremas desarrollado en Francia entre: el INRIA, la *École Polytechnique*, la Universidad de Paris y el CNR. Coq ofrece un lenguaje formal (Gallina) en el que escribir definiciones matemáticas, algoritmos y teoremas para poder obtener pruebas formalmente verificadas. Al desarrollar un sistema crítico como un compilador, en el que el programador confía de manera casi religiosa, es necesario asegurar de manera total la inexistencia de fallos durante la compilación del programa.

La verificación formal nos permite abstraernos del medio a trabajar y tratarlo de una manera general al definirlo mediante la lógica y la matemática. En estos campos podemos abarcar la infinidad de estados y posibilidades y blindarlos ante fallos al definir y demostrar el conjunto de estados en los que se pueden encontrar el compilador.

Coq sigue el estándar de programación funcional, en el que se trata de conseguir una programación *"pura"*. Entendiendo este término como que cada efecto tan solo debe de producir un único resultado. Basada en esta idea aparece de forma muy cercana el concepto de función: Operaciones sobre un dato de entrada que producen, en nuestro caso, tan solo una salida. Una ventaja casi obvia de que se desprende de esto es la facilidad a la hora de entender y razonar. Ya que trabajamos sobre unos conceptos que nos permiten olvidar las peculiaridades del problema a resolver. Si cada operación sobre un dato genera un nuevo tipo de dato, dejando el anterior intacto. No hay necesidad de preocuparse como ese dato es compartido o si puede cambiar parte del dato que genera o si puede quedar corrupto al utilizarlo otro programa.

Coq es un asistente de teoremas, esto es: Un conjunto de herramientas que automatiza algunos de los ámbitos más rutinarios y básicos de la construcción de teoremas a la vez que depende de guía humana para aspectos más difíciles. El núcleo de Coq es un verificador de teoremas muy simple. Que garantiza que tan solo se deducen verdades correctas. Sobre ese núcleo se asienta un entorno que facilita herramientas de alto nivel para la demostración y desarrollo de teoremas, incluyendo librerías básicas con *lemas*, táticas para construir teoremas complejos semiautomáticos y demás.

En los siguiente apartados se tratará de recoger toda la teoría y práctica necesaria para entender las estructuras, teoremas, y demostraciones cubiertas tanto en la implementación de IMP como durante la creación del compilador. Cabe destacar que en las siguiente secciones se tratará de resumir un libro científico sobre los fundamentos software con capacidad lectiva para un curso de grado, por lo tanto se recomienda encarecidamente indagar por cuenta propia en el mismo (que se encuentra en la bibliografía y en las citas) para una comprensión total de los conceptos y de las estructuras típicas de Coq. Para lectores más experimentados se reconocerá que Coq guarda cierta semejanza con sus antecesores como OCaml, ya que nace con la misma idea de ayudar en la investigación de la programación funcional y en la industrialización de software para sistemas críticos.

3.3.2. Tipos inductivos

Los tipos inductivos son los tipos más básicos a definir en Coq. Con ellos definimos tipos con valores o *estados* fijos. Ejemplo:

```
Inductive day : Type :=
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

En este caso hemos definido un tipo inductivo: *day* que tan solo puede tener los valores: *(monday, tuesday, wednesday, thrusday, friday, saturday, sunday)*

3.3.3. Funciones

Siguiendo con el ejemplo anterior, las funciones en Coq se definen tal que:

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday → tuesday
  | tuesday → wednesday
  | wednesday → thursday
  | thursday → friday
  | friday → monday
  | saturday → monday
  | sunday → monday
  end.
```

Con este ejemplo hemos definido una función que opera sobre *day* y nos da como resultado otro tipo *day* indicando el siguiente día de la semana.

Cabe destacar que los argumentos de la función están explicitados en la cabecera de la misma, así como el tipo que devuelve. Coq a veces puede deducir el tipo devuelto cuando no se explicitan.

3.3.4. Tipos

Toda expresión en Coq tiene un tipo asociado que especifica al que hace referencia. el comando *Check* ordena a Coq a mostrar el tipo de una expresión. Ejemplo:

```
Check monday.
  : day.

Check next_weekday
  : day -> day.
```

Funciones como *next_weekday* son también datos con valores. Su tipo se llaman tipos funcionales *(Function types)* y se escriben con flechas.

3.3.5. Tipos recursivos

Los tipos recursivos en Coq quedan definidos con la etiqueta *Fixpoint* tal que:

```
Fixpoint even (n:nat) : bool :=
 match n with
 | O → true
 | S O → false
 | S (S n') → even n'
 end.
```

Para la función recursiva se definen todos los posibles estados como si se tratará de un tipo solo que en uno o más de estos estados aparecerá una llamada al mismo tipo recursivo. En este ejemplo se define el caso base *0 = true* y el caso base *1 = false* para definir el caso recursivo *S(S n') = even n'*

En este caso se ha definido la función par de forma recursiva para la representación de los números en Coq. Dicha representación se explicita a continuación:

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

Por coherencia y comodidad, es necesario analizar esta definición de número tan propia de Coq. Ya que el resto de ejemplos la utilizarán de manera reiterada.

De manera informal, los distintos estados de la definición pueden leerse como:

- O es un número natural (A tener en cuenta que en la definición se encuentra la letra O mayúscula y no el carácter numérico 0)
- S puede situarse enfrente de un número natural para conseguir el siguiente; si n es un número natural, S n también lo es.

De esta forma garantizamos por la construcción de *nat* que todas las expresiones construidas a partir de ella pertenecen al conjunto *nat*

Para acabar se aporta un ejemplo de la construcción recursiva del número 4:

```
Check (S (S (S (S O)))).
    (* ===> 4 : nat *)
```

Partimos de O que sería 0, su siguiente S O que sería el natural 1, su siguiente S (S O) que sería el naturla 2, su siguiente S (S (S O)) que sería el natural 3 y su siguiente S (S (S (S O))) que sería finalmente el natural 4.

3.3.6. Listas

En los tipo inductivos cada constructor puede cero, uno o varios argumentos:

```
Inductive natprod : Type :=
 | pair (n1 n2 : nat).
```

Este tipo puede leerse como:

> "La única forma de construir un par de números es aplicando el constructor *pair* a dos argumentos de tipo *nat*"

Siguiendo esta idea de par de elementos podemos extender la definición al concepto de lista. En el que si se define de forma recursiva, podemos considerarlo como un par formado por, en este caso un número, y una lista:

(n (nat), list (nat))

Por lo tanto y siguiendo esta idea. Definimos en Coq el tipo lista como: una lista vacía o un par formado por un número y una lista:

```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).
```

En la definición podemos observar como el constructor *cons* está formado por dos argumentos. Realmente *cons* es otro nombre para reflejar el mismo concepto que anteriormente se define en *pair* En el siguiente ejemplo se ilustra una lista en Coq formada por tres elementos:

```
Definition mylist := cons 1 (cons 2 (cons 3 nil)).
```

Aquí se ve que la lista está formada de forma recursiva por:

1 + lista (2 + lista (3 + lista vacía))

3.3.7. Polimorfismo

El concepto de polimorfismo abarca la abstracción de las funciones sobre el tipo de dato con el que trabajan. Para ilustrarlo utilizaremos funciones y tipos ya definidos en ejemplos anteriores para facilitar su comprensión y mantener la coherencia.

A continuación se define un tipo polimórfico del tipo lista:

```
Inductive list (X:Type) : Type :=
  | nil
  | cons (x : X) (l : list X).
```

A priori se observa la misma definición que el tipo *natlist* anteriormente definido con la diferencia de que posee un tipo arbitrario X y que las ocurrencias del tipo *nat* han sido reemplazadas por el tipo X.

La X en la definición de la función automáticamente se convierte en un parámetro para el constructor *nil* y *cons* por lo que ahora dichos constructores son polimórficos, a la vez que también lo es la función de la que forman parte. Al comprobar el tipo de *nil* o de *cons* obtenemos:

```
Check nil :  X : Type, list X.
Check cons :  X : Type, X → list X → list X.
```

Gracias a las funcionalidades de Coq podemos hablar de inferencia de tipo *(type inference)*. Coq es capaz de utilizar esto para deducir los tipos de X y de los parámetros que dependen del tipo X. Esto nos permite no tener que escribir explícitamente las anotaciones de tipo en todas las funciones.
Ejemplos:

- *Definición de la función repeat explicitando los tipos.*

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
      | 0 → nil X
      | S count' → cons X x (repeat X x count')
  end.
```

- *Definición de la función repeat recurriendo a la inferencia de tipos de Coq.*

```
Fixpoint repeat' X x count : list X :=
  match count with
      | 0 → nil X
      | S count' → cons X x (repeat' X x count')
  end.
```

Al comprobar los tipos de ambas funciones se pueder ver que Coq obtiene los mismos resultados:

```
Check repeat'
  :  X : Type, X → nat → list X.
Check repeat
  :  X : Type, X → nat → list X.
```

Se introduce también el tipo polimórfico *pair* para acabar de generalizar todos los ejemplos anteriores:

```
Inductive prod (X Y : Type) : Type :=
| pair (x : X) (y : Y).
```

3.3.8. Funciones de alto nivel (higher-order functions)

El concepto de funciones de alto nivel hace referencia a funciones que tienen la capacidad de tratar como datos a otras funciones. Coq trata con especial mimo a estas funciones, permitiéndolas utilizarlas como parámetros de entrada a otras funcionales, devolverlas como resultado, guardarlas en estructuras de datos, etc.

Un ejemplo básico de esto podría ser una función que aplica la función entrada 3 veces:

```
Definition doit3times {X : Type} (f : X→X) (n : X) : X :=
 f (f (f n)).
```

Un ejemplo más interesante de función de alto nivel puede ser la función *Filter*, la cuál dada una condición lógica y una lista, es capaz de filtrar dicha lista en función de la condición:

```
Fixpoint filter {X:Type} (test: X→bool) (l:list X) : list X :=
  match l with
      | [] → []
      | h :: t →
        if test h then h :: (filter test t)
        else filter test t
  end.
```

Al utilizar la función *even* y una lista de números naturales, y pasar ambos elementos como parámetros de entrada a la función *filter* obtenemos otra lista resultado con todos los números pares de la lista originaria:

```
Example test_filter1: filter even [1;2;3;4] = [2;4].
    Proof. reflexivity. Qed.
```

Aclaración: *en el fragmento de código anterior hace aparición el sistema de demostración de teoremas semiautomático de Coq (Proof. reflexivity. Qed) el cual será explicado en el apartado Tácticas*

Otro ejemplo práctica es la función *Map* la cuál tiene como parámetros de entrada una función f y una lista l y devuelve la aplicación de f a cada elemento de la lista l:

```
Fixpoint map {X Y : Type} (f : X→Y) (l : list X) : list Y :=
  match l with
      | [] → []
      | h :: t → (f h) :: (map f t)
  end.
```

Al aplicarla obtenemos:

```
Example test_map1: map (fun x  plus 3 x) [2;0;2] = [5;3;5].
    Proof. reflexivity. Qed.
```

Aclaración: *La función plus (n : nat) (x : nat) : nat no ha sido definida en este proyecto, ni será necesaria. Es una función que suma n unidades al número x. En este caso, suma 3 unidades a cada elemento de la lista.*

Funciones que devuelven funciones

Otros aspecto a tratar son las funciones que devuelven como resultado otra función. Por ejemplo se definirá una función que toma como entrada un valor X y devuelve una función de *nat* a X que devuelve X siempre que se llama, ignorando su parámetro de entrada:

```
Definition constfun {X: Type} (x: X) : nat → X :=
     fun (k:nat)  x.

Definition ftrue := constfun true.

Example constfun_example1 : ftrue 0 = true.
    Proof. reflexivity. Qed.

Example constfun_example2 : (constfun 5) 99 = 5.
    Proof. reflexivity. Qed.
```

3.3.9. Mapas

Los mapas o diccionarios son estructuras de datos que aparecen en cualquier lenguaje de programación, y en nuestro caso serán necesarias para definir los estados de las variables del lenguaje imperativo IMP.

Se definirán a continuación dos tipos de mapas: mapas totales *(total maps)* que incluirán un elemento por defecto a devolver cuando la llave *(key)* que se está buscando no devuelva ningún valor *(value)*, y mapas parciales *(partial maps)* que en su lugar devolverán un opcional *(Optional)* indicando el éxito o el fallo de la búsqueda.

Como introducción, primero se necesita una definición de igualdad para poder comparar las *keys* para ello estableceremos un teorema:

```
Definition eqb_string (x y : string) : bool :=
 if string_dec x y then true else false.
```

Aclaración: *la función string_dec pertenece a la librería estándar de Coq para comparar string de una forma sencilla.*

Al establecer esta definición de igualdad, debemos asegurar ciertas propiedades básicas sobre la misma:

- Reflexividad:

```
Theorem eqb_string_refl :  s : string, true = eqb_string s s.
Proof.
  intros s. unfold eqb_string.
  destruct (string_dec s s) as [Hs_eq | Hs_not_eq].
  - reflexivity.
  - destruct Hs_not_eq. reflexivity.
Qed.
```

- Equivalencia:

```
Theorem eqb_string_true_iff :  x y : string, eqb_string x y = true  x = y.
Proof.
    intros x y.
    unfold eqb_string.
    destruct (string_dec x y) as [Hs_eq | Hs_not_eq].
    - rewrite Hs_eq. split. reflexivity. reflexivity.
    - split.
    + intros contra. discriminate contra.
    + intros H. exfalso. apply Hs_not_eq. apply H.
Qed.
```

Mapas totales (Total Maps)

Para representar mapas se utilizarán funciones en lugar de listas con pares *key-value*. La ventaja de esta elección reside en que dos mapas que responden igual a las consultas serán literalmente la misma función (o estructura de datos) en lugar de ser una equivalencia entre ellas (dos estructuras de datos existencialmente diferentes pero iguales en cuanto a datos).

- Se definirá un mapa total como:

```
Definition total_map (A : Type) :=
    string → A.
```

- Se definirá también un mapa total vacío:

```
Definition t_empty {A : Type} (v : A) : total_map A :=
    (fun _ → v).
```

El cuál devolverá siempre un elemento por defecto *(v : A)* cuando se aplique a cualquier *string* $(fun\_ -> v)$.

Con estas dos definiciones establecidas se puede definir la función más interesante de un mapa; la función *t_update*:

```
Definition t_update {A : Type} (m : total_map A) (x : string) (v : A) :=
    fun x' → if eqb_string x x' then v else m x'.
```

Este es un ejemplo claro de función de alto nivel; *t_update* necesita de un mapa m, una *key* x y un *value* v, y devuelve un nuevo mapa que asigna x a v y el resto de *key* las deja igual que el mapa de entrada.

A modo de ejemplo se creará un mapa total que asigne *booleans* a *strings*, donde "foo"y "bar", están mapeados a *true* y cualquier otra *key* está mapeada a *false.* Ejemplo:

```
Definition examplemap :=
 t_update (t_update (t_empty false) "foo" true) "bar" true.
```

Se introduce ahora un concepto de Coq meramente estético: el concepto de notaciones. Las notaciones en Coq sirven el propósito de simplificar la lectura del código e incluso conseguir cierta identidad estética característica de estructuras de datos, tipos de datos, formas, etc...

En este caso se introducirá una notación para los mapas totales que hará más llevadera su interpretación y lectura:

```
Notation "x '!->' v ';' m" := (t_update m x v)
    (at level 100, v at next level, right associativity).
```

Nota: *Para más información acerca de cómo se construyen las anotaciones en Coq se recomienda el manual y el libro citado en la bibliografía.)*

Para ilustrar este concepto se muestra el ejemplo anterior de mapa total con la pertinente notación recién creada:

```
Definition examplemap' :=
  ( "bar" !-> true;
    "foo" !-> true;
    _ !-> false).
```

Mapas parciales (Partial Maps)

Los mapas parciales son mapas totales con la única diferencia de que los elementos que lo forman son del tipo *option* A y por defecto el elemento *None*.

```
Definition partial_map (A : Type) := total_map (option A).

Definition empty {A : Type} : partial_map A :=
    t_empty None.

Definition update {A : Type} (m : partial_map A) (x : string) (v : A) :=
    (x !-> Some v ; m).
```

Definimos también notaciones para el mapa parcial:

```
Notation "x '>' v ';' m" := (update m x v)
(at level 100, v at next level, right associativity).

Notation "x '>' v" := (update empty x v)
(at level 100).
```

Y lo ejemplificamos:

```
Example examplepmap :=
 ("Church" > true ; "Turing" > false).
```

Aclaración: *Se expandirá ahora el tipo Option y None utilizados en la definición de los mapas parciales*

3.3.10. Tipo Option

Este tipo utilizado en el ejemplo anterior sirve el propósito de capturar mensajes de error en Coq al devolver valores no esperados o nulos. Su implementación para, por ejemplo, el tipo *nat* es la siguiente:

```
Inductive natoption : Type :=
  | Some (n : nat)
  | None.
```

Un ejemplo de su uso sería la creación de un método *nth_error* que devuelve el número en la posición indicada, si no puede acceder a dicha posición devuelve *None*

```
Fixpoint nth_error (l:natlist) (n:nat) : natoption :=
match l with
| nil → None
| a :: l' → match n with
           | O → Some a
           | S n' → nth_error l' n'
           end
end.
Example test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.
    Proof. reflexivity. Qed.
Example test_nth_error2 : nth_error [4;5;6;7] 3 = Some 7.
    Proof. reflexivity. Qed.
Example test_nth_error3 : nth_error [4;5;6;7] 9 = None.
    Proof. reflexivity. Qed.
```

Aprovechando el poder del polimorfismo se puede implementar un tipo *Option* que sirva para cualquier tipo:

```
Inductive option (X:Type) : Type :=
    | Some (x : X)
    | None.
```

3.3.11. Tácticas

Este subcapítulo de teoría de Coq sirve más bien como recordatorio del aviso de ampliar los conocimientos expuestos en este proyecto. Ya que para la parte práctica de formalización de un compilador y demostración de sus propiedades se hará un uso intensivo de las distintas tácticas y estrategias para la demostración de teoremas.

3.4– Teoría de lenguajes de programación

3.4.1. Sintaxis

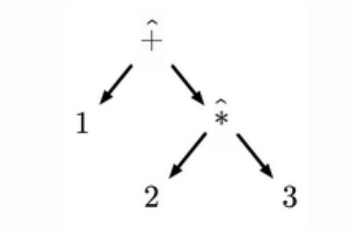
Los lenguajes de programación empieza con primitivas u objetos atómicos que representan unidades de significado. La aritmética tiene solo una primitiva: los números. Mientras que los lenguajes de programación tienen muchas primitivas como caracteres, booleanos, funciones y demás.

El objetivo de un lenguaje es definir estructuras sobre las primitivas básicas mediante composición de las mismas, entendiendo la composición como cualquier operación o estructura que use o relacione estás primitivas. Definimos ahora estas estructuras de composición para la aritmética con una gramática libre de contexto:

$$\begin{aligned} \textbf{Number } n &\in \mathbb{N} \\ \textbf{Binop } \hat{\oplus} &::= \hat{+} \mid \hat{-} \mid \hat{*} \mid \hat{/} \\ \textbf{Expression } e &::= n \mid e_1 \mathbin{\hat{\oplus}} e_2 \end{aligned}$$

3.4.2. Árboles sintácticos

Cuando se escribe en texto una expresión aritmética esta se observa como una sucesión de caracteres, ejemplo: 1 + 2 * 3. Pero esta impresión es fruto de la limitación del medio textual. Las gramáticas libres de contexto describen estructuras de tipo árbol; el carácter lineal del texto anterior describe implícitamente este árbol sintáctico:



Con esta idea podemos definir la sintaxis como la estructura básica de un lenguaje. El árbol correspondiente a una definición sintáctica es llamado árbol sintáctico abstracto, o AST *(Abstract Syntax Tree)*.

3.4.3. Semántica

La sintaxis define el conjunto de formas permitidas en nuestro lenguaje, la estructura de nuestro dominio. El siguiente paso a definir en un lenguaje de programación sería establecer su semántica y propiedades. En el caso de la aritmética, el campo que estudia como hablamos sobre expresiones aritméticas, o incluso se cuestiona su veracidad y correctitud.

En teoría de lenguajes de programación se busca más una noción de computación como semántica; la meta no es asignar semántica al *string* "hello" sino dejar al programador decidir como interpretar el contenido de un *string* en función de sus necesidades. Por ejemplo, un programa es una estructura que puede ser computada: "hello" + "world" se puede recudir a "hello world". De manera más general:

> "Dado un lenguaje que define expresiones, la meta es definir una semántica que pueda reducir esas expresiones a sus formas primitivas"

3.4.4. Operaciones semánticas

Se define una noción formal de computación para expresiones del lenguaje a través de semánticas *small-steps*. Para cualquier expresión e puede estar en uno de los dos estados siguientes: Reducible, lo cual quiere decir que se puede usar operaciones semánticas para reducirla, o la expresión es un valor, entendiendo esto como que es irreducible y ha alcanzado su forma final. Se formaliza estos estados como enunciados lógicos. escritos como $e -> e'$ para hacer referencia a "***e pasa a e'***" y e val para "***e es un valor***" . Se define ahora un sistema formal para demostrar estos enunciados sobre expresiones; las operaciones semánticas para la aritmética:

$$\frac{}{n\ \mathsf{val}}\ \text{(D-Num)} \qquad \frac{e_1 \mapsto e_1'}{e_1 \mathbin{\hat{\oplus}} e_2 \mapsto e_1' \mathbin{\hat{\oplus}} e_2}\ \text{(D-Left)} \qquad \frac{e_1\ \mathsf{val} \quad e_2 \mapsto e_2'}{e_1 \mathbin{\hat{\oplus}} e_2 \mapsto e_1 \mathbin{\hat{\oplus}} e_2'}\ \text{(D-Right)} \qquad \frac{n_1 \oplus n_2 = n_3}{n_1 \mathbin{\hat{\oplus}} n_2 \mapsto n_3}\ \text{(D-}\oplus\text{)}$$

Cada una de estas operaciones representa una regla de inferencia, las cuales se pueden leer como implicaciones verticales $\frac{a}{b}$ es lo mismo que a $\rightarrow$ b, o "Si lo superior es cierto, por implicación lógica, lo inferior también".

3.4.5. Demostración por reducción de términos

Para ver el potencial de las reglas de inferencias definidas anteriormente, se demostrará una reducción bastante simple: $1 + 6 * 3/2$ es igual a 10. Se usará la sintaxis $e_1 \rightarrow e_2$ para simbolizar que: "e_1 se reduce a e_2 después de cero o más estados'

, por lo tanto nuestra meta es demostrar que $1 + 6 * 3/2 \rightarrow 10$. La evaluación completa tiene demasiados pasos, con solo demostrar la primera reducción será suficiente para reflejar la metodología. Siguiendo nuestras reglas de inferencia u operaciones semánticas, se empezará por: $e_1 = 1, e_2 = 6 * 3/2$:

$$\frac{}{1 \mathbin{\hat{+}} 6 \mathbin{\hat{*}} 3 \mathbin{\hat{/}} 2 \mapsto ?}\ (?)$$

Esta imagen se podría leer como: *"La meta a demostrar es que la expresión se reduce a otra expresión. Pero no se sabe con certeza que regla o reducción aplicar"*. La estrategia de demostración aquí es sencillamente una búsqueda por fuerza bruta. Hay cuatro posibles reglas que se podrían aplicar a la expresión, por lo que habrá que estudiarlas para ver cual es la más propicia:

- D-Num: La expresión no es un número, por lo que no es aplicable.
- D-Left: Esta regla se aplica si e_1 puede reducirse. $e_1 = 1$ por lo tanto e_1 val, así que tampoco es aplicable.
- D-$\bigoplus$: Esta regla se aplica si e_1 y e_2 son números. e_2 no es un número por lo que tampoco es aplicable.
- D-Right: Esta regla se aplica si e_1 es un valor, y e_2 un expresión reducible. Esta regla parece ser la indicada para esta expresión:

$$\frac{\dfrac{}{1 \text{ val}}\ (\text{D-Num}) \qquad \dfrac{}{6 \mathbin{\hat{*}} 3 \mathbin{\hat{/}} 2 \mapsto ?}\ (?)}{1 \mathbin{\hat{+}} 6 \mathbin{\hat{*}} 3 \mathbin{\hat{/}} 2 \mapsto ?}\ (\text{D-Right})$$

Siguiendo este proceso y aplicando distintas reducciones de expresiones se puede obtener toda la demostración de la expresión final. Dada una meta, recursivamente se demuestran las asunciones hasta que alcancemos axiomas o expresiones atómicas. Una vez que todas las expresiones están correctamente reducidas se puede hablar de que la expresión esta completamente demostrada. Aplicando reiteradamente la estrategia de demostración mencionada arriba se llegará a dicha demostración:

$$\frac{\dfrac{}{1 \text{ val}}\ (\text{D-Num}) \qquad \dfrac{\dfrac{\dfrac{}{6 * 3 = 18}\ (\text{arithmetic})}{6 \mathbin{\hat{*}} 3 \mapsto 18}\ (\text{D-}\oplus)}{6 \mathbin{\hat{*}} 3 \mathbin{\hat{/}} 2 \mapsto 18 \mathbin{\hat{/}} 2}\ (\text{D-Left})}{1 \mathbin{\hat{+}} 6 \mathbin{\hat{*}} 3 \mathbin{\hat{/}} 2 \mapsto 1 \mathbin{\hat{+}} 18 \mathbin{\hat{/}} 2}\ (\text{D-Right})$$

3.5– IMP

3.5.1. Introducción

En estos capítulos se definirá todas las partes necesarias del lenguaje de programación imperativa IMP el cual nos servirá como base para corroborar semánticamente que el compilador mantiene el significado de sus expresiones al traducirlo.

IMP está basado en un pequeño fragmento de lenguajes de programación tan convencionales como C y Java. Es un lenguaje aritmético básico que se irá extendiendo poco a poco hasta abarcar variables y sus estados.

También, en estas secciones se verá como IMP puede ser evaluado desde un punto de vista funcional, tratándolo como una síntesis y composición de funciones, o estudiando las relaciones entre sus elementos más básicos.

3.5.2. Sintaxis

Se aportará la definición sintáctica de forma abstracta de las expresiones aritméticas y booleanas:

Definición aritmética:

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).
```

Se definen las estructuras básicas aritméticas de:

- ANum (n : nat): Primitiva básica, expresión atómica formada por un único valor, un número de tipo *nat*.
- Aplus (a1 a2 : aexp): Expresión suma, estructura formada por dos expresiones reducibles del tipo *aexp*.
- AMinus (a1 a2 : aexp):Expresión resta, estructura formada por dos expresiones reducibles del tipo *aexp*.
- AMult (a1 a2 : aexp):Expresión multiplicación, estructura formada por dos expresiones reducibles del tipo *aexp*.

Como se puede observar, el tipo *aexp* es auto inclusivo ya que en su misma definición se utiliza para definir estados posibles. Lo que posteriormente le impregnará el carácter recursivo a la hora de reducir las expresiones del mismo tipo hasta conseguir el tipo (o estado) atómico ANum.

Para conocer como esta definición sintáctica abstracta (o árbol sintáctico abstracto) se transcribiría a una sintaxis concreta se necesitaría de un analizador y *parseador* léxico que realice esta traducción. Es decir, cómo de la expresión concreta: $1 + 2 * 3$ se obtendría la expresión abstracta: APlus (ANum 1) (AMult (ANum 2) (ANum 3)).

Para comprender este concepto de abstracción, y como puede expresarse sintácticamente lo mismo pero con otra forma, se introduce una definición gramática convencional de la misma definición abstracta anterior en forma BNF (Backus-Naur Form):

```
a := nat
| a + a
| a - a
| a × a
```

Comparado con la definición en Coq aportada más arriba:

- La forma BNF es más informal; da pistas sobre la forma sintáctica de las expresiones, dejando otros aspectos como el análisis léxico y el *parseo* sin especificar. Además de que sirve de manera únicamente informativa. Para poder definir un compilador se necesitaría formalizarlo.
- Por consiguiente, la versión BNF es mucho más entendible a la hora de ser leída. Esta informalidad le da más flexibilidad a la definición. Por lo tanto a la hora de ser discutida y explicada es mucho más conveniente.

Definición booleana:

```
Inductive bexp : Type :=
    | BTrue
    | BFalse
    | BEq (a1 a2 : aexp)
    | BLe (a1 a2 : aexp)
    | BNot (b : bexp)
    | BAnd (b1 b2 : bexp).
```

Se definen las estructuras básicas booleanas de:

- BTrue: Primitiva básica, expresión atómica formada por un único valor, un *boolean* True.
- BFalse: Primitiva básica, expresión atómica formada por un único valor, un *boolean* False.
- BEq (a1 a2 : aexp): Expresión *Equal*, estructura formada por dos expresiones reducibles del tipo *bexp*.
- BLe (a1 a2 : aexp): Expresión *Less than or equal*, estructura formada por dos expresiones reducibles del tipo *bexp*.
- BNot (b : bexp): Expresión *Not*, estructura formada por una expresión reducible del tipo *bexp*.
- BAnd (b1 b2 : bexp): Expresión *And*, estructura formada por dos expresiones reducibles del tipo *bexp*.

Como se puede observar se guarda cierta similitud con la definición sintáctica aritmética definida anteriormente. Sin embargo, se denotan sutiles diferencias:

- Existen dos tipos atómicos que no toman valores de entrada. A diferencia de ANum el cual tomaba un valor del tipo *nat*, estos tipos solo pueden tomar los valores de BTrue y BFalse.
- Existen tres estructuras sintácticas compuestas por dos expresiones reducibles. Las cuales a diferencia de las aritméticas tienen funciones distintas: Igualdad lógica, Menor o igual que y la operación y. *(Equal, Less than or equal y And, respectivamente)*.
- Existe una estructura sintáctica diferente a las anteriores, la estructura *BNot* la cual está formada tan solo por una única expresión reducible.

Al igual que antes, definimos una nueva abstracción sintáctica de forma BNF para la expresión booleanas. Cabe recordar que al estar tratando con la sintaxis abstracta se describe el conjunto de estructuras básicas del lenguaje, y no la forma en la que estas se transcribirían al ser concretizadas. Esto se conoce como el léxico del lenguajes; el cómo se escribe. Dicha definición sería:

```
 b := true
| false
| a = a
| a  a
| ¬b
| b && b
```

3.5.3. Evaluación como función

Evaluar una expresión aritmética produce un número:

```
Fixpoint aeval (a : aexp) : nat :=
 match a with
     | ANum n  n
     | APlus a1 a2  (aeval a1) + (aeval a2)
     | AMinus a1 a2  (aeval a1) - (aeval a2)
     | AMult a1 a2  (aeval a1) × (aeval a2)
 end.

Example test_aeval1: aeval (APlus (ANum 2) (ANum 2)) = 4.
   Proof. reflexivity. Qed.
```

De igual manera, evaluar una expresión booleana produce un valor booleano:

```
Fixpoint beval (b : bexp) : bool :=
 match b with
     | BTrue  true
     | BFalse  false
     | BEq a1 a2  (aeval a1) =? (aeval a2)
     | BLe a1 a2  (aeval a1) <=? (aeval a2)
     | BNot b1  negb (beval b1)
     | BAnd b1 b2  andb (beval b1) (beval b2)
 end.

 Example test_beval1: aeval (AEq (BTrue) (BTrue)) = true.
   Proof. reflexivity. Qed.
```

3.5.4. Evaluación como relación

Se ha presentado *aeval* y *beval* como funciones definidas por *Fixpoints*. Otra forma de ver la evaluación sería como una relación entre las expresiones y sus valores. Esto implica necesariamente a la formación de estructuras y tipos recursivos.

Se aporta a continuación la misma evaluación aritmética que antes pero con un punto de vista relacional:

```
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      aevalR (ANum n) n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (AMult e1 e2) (n1 × n2).
```

Para "facilitar"su lectura, podemos añadir nombres a las hipótesis para cada regla de inferencia (u operación semántica):

```
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      aevalR (ANum n) n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat)
      (H1 : aevalR e1 n1)
      (H2 : aevalR e2 n2) :
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat)
      (H1 : aevalR e1 n1)
      (H2 : aevalR e2 n2) :
      aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat)
      (H1 : aevalR e1 n1)
      (H2 : aevalR e2 n2) :
      aevalR (AMult e1 e2) (n1 × n2).
```

De esta manera al utilizar el demostrador de teoremas estos nombres quedarán fijados a las hipótesis. Es un precio a pagar a pesar de añadir más código a la definición inductiva.

Para ahora si, facilitar su comprensión y lectura en futuros ejemplos se añadirá una notación para *aevalR*. Al escribir e ==> n se hará referencia a que la expresión aritmética e se evalúa como el valor n.

```
Notation "e '==>' n"
 := (aevalR e n)
    (at level 90, left associativity)
 : type_scope.
```

Se puede utilizar esta nueva notación para volver a definir *aevalR* de una manera aun más gráfica:

```
Reserved Notation "e '==>' n" (at level 90, left associativity).
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      (ANum n) ==> n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) → (e2 ==> n2) → (APlus e1 e2) ==> (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) → (e2 ==> n2) → (AMinus e1 e2) ==> (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) → (e2 ==> n2) → (AMult e1 e2) ==> (n1 × n2)

  where "e '==>' n" := (aevalR e n) : type_scope.
```

3.5.5. Equivalencia de las definiciones

Se intuye de la equivalencia entre ambas formas de definir la evaluación de las expresiones aritméticas, pero a continuación se explicita su deducción lógica:

```
Theorem aeval_iff_aevalR' :  a n,
(a ==> n)  aeval a = n.
    Proof.
      split.
      - (* -> *)
        intros H; induction H; subst; reflexivity.
      - (* <- *)
        generalize dependent n.
        induction a; simpl; intros; subst; constructor;
           try apply IHa1; try apply IHa2; reflexivity.
    Qed.
```

Esta demostración encierra un significado muy potente. Ya que utilizando las herramientas explicadas durante este proyecto (programación funcional, verificación formal y definiciones matemáticas) se puede asegurar que la totalidad de expresiones aritméticas evaluadas tanto con una definición funcional (O computacional) como relacional son equivalentes.

Debido a que todos los elementos han sido definidos matemáticamente, podemos abstraer esta demostración y ser capaces de generalizar conocimiento, demostraciones y pruebas evadiendo la ardua tarea de corroborar todas y cada una de, en este caso, concretizaciones de expresiones aritméticas. Es decir, para alcanzar este nivel de seguridad en un enunciado se debería de probar para todo el conjunto de posibles expresiones aritméticas, y se deberían de evaluar con ambas definiciones para, efectivamente, llegar a la conclusión total de que son equivalentes.

Esta es la potencia de las definiciones lógicas y de la verificación formal de las mismas.

Para las definiciones aportadas, todo es cuestión del caso a tratar. Por ejemplo, si se quiere extender la definición aritmética con una nueva operación, la definición relacional es mucho más cómoda:

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp)
  | ADiv (a1 a2 : aexp). (* <--- NUEVO: división *)
```

También se necesita ampliar la definición relacional de las expresiones aritméticas para incluir esta nueva operación:

```
Reserved Notation "e '==>' n"
          (at level 90, left associativity).

Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      (ANum n) ==> n
  | E_APlus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (APlus a1 a2) ==> (n1 + n2)
  | E_AMinus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMinus a1 a2) ==> (n1 - n2)
  | E_AMult (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMult a1 a2) ==> (n1 × n2)
  | E_ADiv (a1 a2 : aexp) (n1 n2 n3 : nat) : (* <----- NUEVO: evaluación*)
      (a1 ==> n1) → (a2 ==> n2) → (n2 > 0) →
      (mult n2 n3 = n1) → (ADiv a1 a2) ==> n3

where "a '==>' n" := (aevalR a n) : type_scope.
```

Introduciendo esta nueva operación, la definición *aevalR* es ahora una definición parcial, ya que hay algunos valores de entrada (n1 y n2 en la definición de división) que no intervienen de manera directa en los valores de salida.

Un punto fuerte de las definiciones relacionales es que son más elegantes y fáciles de entender.
Sin embargo, la definición funcional puede ser más conveniente para:

- Las funciones son deterministas por definición y definidas para todos sus argumentos; en el caso de las relaciones estas propiedades tienen que ser demostradas de forma explicitas cuando se necesiten.
- Con las funciones podemos aprovechar el potencial computacional de Coq para simplificar las expresiones durante las demostraciones.

3.5.6. Estados y variables

El siguiente paso a definir en IMP es añadir riqueza y complejidad a las expresiones mediante variables. Para no escalar mucho la complejidad del proyecto, se suponen variables globales y que solo almacenan números.

Como se necesita almacenar el valor de las variables, el tipo Mapa definido en su correspondiente apartado teórico será lo propio para ello.
Las variables serán por lo tanto *string* que estarán vinculados a sus valores de tipo *nat*.

Un estado máquina *(machine state)* (o simplemente estado) se representa como el conjunto de los valores de las variables en un momento exacto de la ejecución de un programa.

Por simplicidad, se asumirá que el estado está definido para todo el conjunto de variables, incluso si el programa a tratar solo utiliza un número finito de ellas. Este estado captura toda la información almacenada en memoria. Para nuestros programas escritos en IMP, como cada variable almacena un número natural, el estado se puede representar como una mapa de *string* a *nat* y se usará el 0 como valor por defecto a almacenar. Esta es un simplificación de cómo se tratan los estados en la realidad, ya que los lenguajes de programación modernos son más complejos.

```
Definition state := total_map nat.
```

Sintaxis de las variables

Se añadirá ahora el concepto de variable a la definición aritmética añadiendo tan solo un nuevo constructor:

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | AId (x : string) (* <--- NUEVO: variables *)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).

Definition W : string := "W".
Definition X : string := "X".
Definition Y : string := "Y".
Definition Z : string := "Z".
```

Se han definido también algunos nombres de variables como notaciones que harán los ejemplos más fáciles de leer y entender.

En cuánto a la definición de las expresiones booleanas, esta queda igual debido a que los únicos valores que puede tomar *(true y false)* ya están definidos como constructores *(BTrue y BFalse)*.

Para facilitar la comprensión de IMP y para que su código siga las convenciones típicas de los lenguajes, se define un conjunto de notaciones para sus operaciones aritméticas y booleanas:

```
Declare Scope com_scope.
 Notation "<{ e }>" := e (at level 0, e custom com at level 99) : com_scope.
 Notation "( x )" := x (in custom com, x at level 99) : com_scope.
 Notation "x" := x (in custom com at level 0, x constr at level 0) : com_scope.
 Notation "f x .. y" := (.. (f x) .. y)
                  (in custom com at level 0, only parsing,
                  f constr at level 0, x constr at level 9,
                  y constr at level 9) : com_scope.
 Notation "x + y" := (APlus x y) (in custom com at level 50, left associativity).
 Notation "x - y" := (AMinus x y) (in custom com at level 50, left associativity).
 Notation "x * y" := (AMult x y) (in custom com at level 40, left associativity).
 Notation "'true'" := true (at level 1).
 Notation "'true'" := BTrue (in custom com at level 0).
 Notation "'false'" := false (at level 1).
 Notation "'false'" := BFalse (in custom com at level 0).
 Notation "x <= y" := (BLe x y) (in custom com at level 70, no associativity).
 Notation "x = y" := (BEq x y) (in custom com at level 70, no associativity).
 Notation "x && y" := (BAnd x y) (in custom com at level 80, left associativity).
 Notation "'~' b" := (BNot b) (in custom com at level 75, right associativity).
Open Scope com_scope.
```

Evaluación de las variables

Para evaluar las expresiones aritméticas y booleanas con variables se necesita introducir como parámetro de entrada un estado, lo cual es lo novedoso en esta definición:

```
Fixpoint aeval (st : state) (a : aexp) : nat :=
match a with
    | ANum n  n
    | AId x  st x (* <--- NUEVO: variables *)
    | <{a1 + a2}>  (aeval st a1) + (aeval st a2)
    | <{a1 - a2}>  (aeval st a1) - (aeval st a2)
    | <{a1 × a2}>  (aeval st a1) × (aeval st a2)
end.

Fixpoint beval (st : state) (b : bexp) : bool :=
match b with
    | <{true}>  true
    | <{false}>  false
    | <{a1 = a2}>  (aeval st a1) =? (aeval st a2)
    | <{a1  a2}>  (aeval st a1) <=? (aeval st a2)
    | <{¬ b1}>  negb (beval st b1)
    | <{b1 && b2}>  andb (beval st b1) (beval st b2)
end.
```

También será necesario explicitar que para el estado vacío (mapa vacío), el valor a añadir por defecto es el 0:

```
Definition empty_st := (_ !-> 0).
```

Por último se añade la notación necesaria para asignar valores a variables y se exponen ejemplos:

```
Notation "x '!->' v" := (x !-> v ; empty_st) (at level 100).

Example aexp1 :
    aeval (X !-> 5) <{ 3 + (X × 2) }>
  = 13.
Proof. reflexivity. Qed.

Example aexp2 :
    aeval (X !-> 5 ; Y !-> 4) <{ Z + (X × Y) }>
  = 20.
Proof. reflexivity. Qed.

Example bexp1 :
    beval (X !-> 5) <{ true && ¬(X  4) }>
  = true.
Proof. reflexivity. Qed.
```

3.5.7. Comandos

Se definirá ahora la sintaxis y la evaluación de los comandos en IMP.

Sintaxis de los comandos

De manera informal, los comandos c siguen la siguiente descripción según la gramática BNF:

```
c := skip | x := a | c ; c | if b then c else c end
| while b do c end
```

Lo que se define de manera formal como:

```
Inductive com : Type :=
 | CSkip
 | CAsgn (x : string) (a : aexp)
 | CSeq (c1 c2 : com)
 | CIf (b : bexp) (c1 c2 : com)
 | CWhile (b : bexp) (c : com).
```

Se puede definir también algunas notaciones para hacer más fácil la lectura y escritura de programas en IMP:

```
Notation "'skip'" :=
CSkip (in custom com at level 0) : com_scope.
Notation "x := y" :=
     (CAsgn x y)
        (in custom com at level 0, x constr at level 0,
         y at level 85, no associativity) : com_scope.
Notation "x ; y" :=
     (CSeq x y)
       (in custom com at level 90, right associativity) : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" :=
     (CIf x y z)
       (in custom com at level 89, x at level 99,
        y at level 99, z at level 99) : com_scope.
Notation "'while' x 'do' y 'end'" :=
     (CWhile x y)
        (in custom com at level 89, x at level 99, y at level 99) : com_scope.
```

Por ejemplo, se presenta ahora la definición de factorial escrita con las notaciones y las estructuras creadas hasta el momento. Especial hincapié en la aparición de los comandos mediante anotaciones (; while):

```
Definition fact_in_coq : com :=
 <{ Z := X;
    Y := 1;
    while ¬(Z = 0) do
      Y := Y × Z;
      Z := Z - 1
    end }>.
Print fact_in_coq.
```

Evaluación de comandos

El problema aquí es que el bucle *while* puede derivar en estados no conclusos, por lo tanto evaluarlo como función nos llevará a error. Por lo tanto, se procederá directamente a definirlo de manera relacional:

```
Reserved Notation
 "st '=[' c ']=>' st'"
 (at level 40, c custom com at level 99,
  st constr, st' constr at next level).

Inductive ceval : com → state → state → Prop :=
  | E_Skip :  st,
      st =[ skip ]=> st
  | E_Asgn :  st a n x,
      aeval st a = n →
      st =[ x := a ]=> (x !-> n ; st)
  | E_Seq :  c1 c2 st st' st'',
      st =[ c1 ]=> st' →
      st' =[ c2 ]=> st'' →
      st =[ c1 ; c2 ]=> st''
  | E_IfTrue :  st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]=> st' →
      st =[ if b then c1 else c2 end]=> st'
  | E_IfFalse :  st st' b c1 c2,
      beval st b = false →
      st =[ c2 ]=> st' →
      st =[ if b then c1 else c2 end]=> st'
  | E_WhileFalse :  b st c,
      beval st b = false →
      st =[ while b do c end ]=> st
  | E_WhileTrue :  st st' st'' b c,
      beval st b = true →
      st =[ c ]=> st' →
      st' =[ while b do c end ]=> st'' →
      st =[ while b do c end ]=> st''

where "st =[ c ]=> st'" := (ceval c st st').
```

Esta sería la definición formal de la evaluación como función de los estados. Esta definición se corresponde con las siguientes operaciones semánticas (o reglas de inferencia):

$$\frac{}{st =[\ \texttt{skip}\ ]\Rightarrow st}\ (E\_Skip)$$

$$\frac{aeval\ st\ a = n}{st =[\ x := a\ ]\Rightarrow (x\ !\to n\ ;\ st)}\ (E\_Asgn)$$

$$\frac{st =[\ c_1\ ]\Rightarrow st' \quad st' =[\ c_2\ ]\Rightarrow st''}{st =[\ c_1;c_2\ ]\Rightarrow st''}\ (E\_Seq)$$

$$\frac{beval\ st\ b = true \quad st =[\ c_1\ ]\Rightarrow st'}{st =[\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}\ ]\Rightarrow st'}\ (E\_IfTrue)$$

$$\frac{beval\ st\ b = false \quad st =[\ c_2\ ]\Rightarrow st'}{st =[\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{end}\ ]\Rightarrow st'}\ (E\_IfFalse)$$

$$\frac{beval\ st\ b = false}{st =[\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}\ ]\Rightarrow st}\ (E\_WhileFalse)$$

$$\frac{beval\ st\ b = true \quad st =[\ c\ ]\Rightarrow st' \quad st' =[\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}\ ]\Rightarrow st''}{st =[\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}\ ]\Rightarrow st''}\ (E\_WhileTrue)$$

Se usará la notación $st = [c] => st'$ que significa: ejecutar el programa c en un estado inicial st resultará en el estado final st

. Esto puede ser acortado cómo: “c lleva el estado st a st’. ”

Determinismo de la evaluación

El cambiar de una definición funcional a una relacional permite evitar la restricción artificial de que la evaluación tenga que ser una función total. Esta restricción viene dada debido a que Coq no solo es un lenguaje de programación funcional, sino que también es un sistema de evaluación lógica consistente, y cualquier función que no garantice que llegará a un estado final tiene que ser rechazada.

Pero si la evaluación funcional es una parcial, ¿No lo sería también la evaluación relacional, al ser equivalentes? Si esto fuera así, partiendo del mismo estado inicial st, se podría evaluar ciertos comandos c de diferentes maneras para alcanzar dos estados finales distintos, st’y st”.

De hecho, se demuestra que esto no puede pasar. Por lo tanto *ceval* es una función parcial.

```
      Theorem ceval_deterministic:  c st st1 st2,
    st =[ c ]=> st1 →
    st =[ c ]=> st2 →
    st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2.
  induction E1; intros st2 E2; inversion E2; subst.
  - (* E_Skip *) reflexivity.
  - (* E_Asgn *) reflexivity.
  - (* E_Seq *)
    rewrite (IHE1_1 st'0 H1) in ×.
    apply IHE1_2. assumption.
  - (* E_IfTrue, b se evalúa como true *)
      apply IHE1. assumption.
  - (* E_IfTrue,  b se evalúa como false (contradicción) *)
      rewrite H in H5. discriminate.
  - (* E_IfFalse, b se evalúa como true (contradicción) *)
      rewrite H in H5. discriminate.
  - (* E_IfFalse, b se evalúa como false *)
      apply IHE1. assumption.
  - (* E_WhileFalse, b se evalúa como false *)
    reflexivity.
  - (* E_WhileFalse, b se evalúa como true (contradicción) *)
    rewrite H in H2. discriminate.
  - (* E_WhileTrue, b se evalúa como false (contradicción) *)
    rewrite H in H4. discriminate.
  - (* E_WhileTrue, b se evalúa como true *)
    rewrite (IHE1_1 st'0 H3) in ×.
    apply IHE1_2. assumption. Qed.
```

CAPÍTULO 4

Aportación realizada

4.1– Introducción

Una vez asentadas todas las bases y fundamentos teóricos necesarios se procede a la creación de un compilador para el lenguaje IMP definido en su correspondiente capítulo. Se ha decidido exponer dicho caso práctico para corroborar que las bases para construir un lenguaje de programación mediante su formalización funcional y matemática se mantienen incluso si cambiamos su concretización.

Es decir, se han explicitado unas reglas sintácticas y semánticas para la definición de un lenguaje en concreto (IMP). Ahora se plantea cambiar esa concretización y comprobar que a nivel semántico sigue funcionando de manera equivalente. Para ello se implementará un compilador que traducirá el código en IMP a instrucciones en una máquina virtual. Las abstracciones semánticas deben regirse igual en ambos medios.

De manera introductoria se presenta un poco de contexto sobre las tecnologías a usar y tecnologías similares que, a menudo, se mencionan de manera indiscriminada evidenciando un desconocimiento general de las mismas:

Un programa computacional por definición hace referencia a un conjunto de instrucciones escritas en un lenguaje de programación las cuales un ordenador puede ejecutar o interpretar. Aquí se introducen dos conceptos importantes: ejecutar o interpretar.

En realidad, interpretar un programa es una forma de ejecución del mismo. Se entiende como ejecución de un programa el proceso por el cual un ordenador o máquina virtual lee y realiza una instrucción de un programa informático. Cada instrucción de un programa describe una acción en particular que debe ser realizada para resolver un problema en concreto.

La diferencia real surge entre los términos compilado o interpretado. Se dice que un programa/lenguaje de programación es interpretado cuando las instrucciones se ejecutan de manera inmediata. Sin la necesidad de que el código tenga que ser compilado o traducido a otro lenguaje.

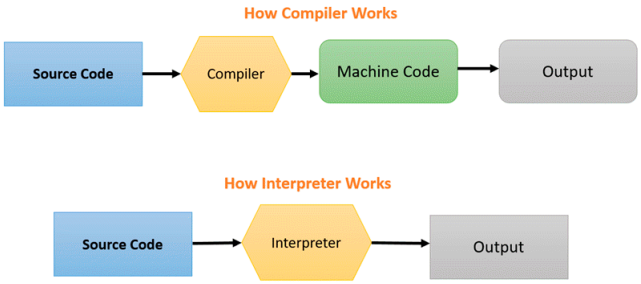


Figura 4.1: Esquemas de ejecución

Mientras que un lenguaje/programa es compilado cuando se requiere que cada instrucción sea traducida, o bien a lenguaje máquina para ser ejecutada directamente, o bien a otro lenguaje de programación generalmente de más bajo nivel que el lenguaje original.

En el caso particular de este proyecto, las instrucciones IMP serán traducidas a un conjunto de instrucciones propio de una máquina virtual. Una máquina virtual es una virtualización o emulación de un sistema informático. El por qué de la utilización de máquinas virtuales para ejecutar programas viene motivado por la portabilidad y capacidad de abstracción frente al sistema operativo.

Si no se utilizarán maquinas virtuales, los programas tendrían que ser escritos de manera especifica a cada sistema operativo para ser ejecutados. De esta manera, los programas son escritos para ser compilados y ejecutados por la máquina virtual. Lo que permite que solo estos últimos tengan que adaptarse al sistema operativo en el que se encuentran.

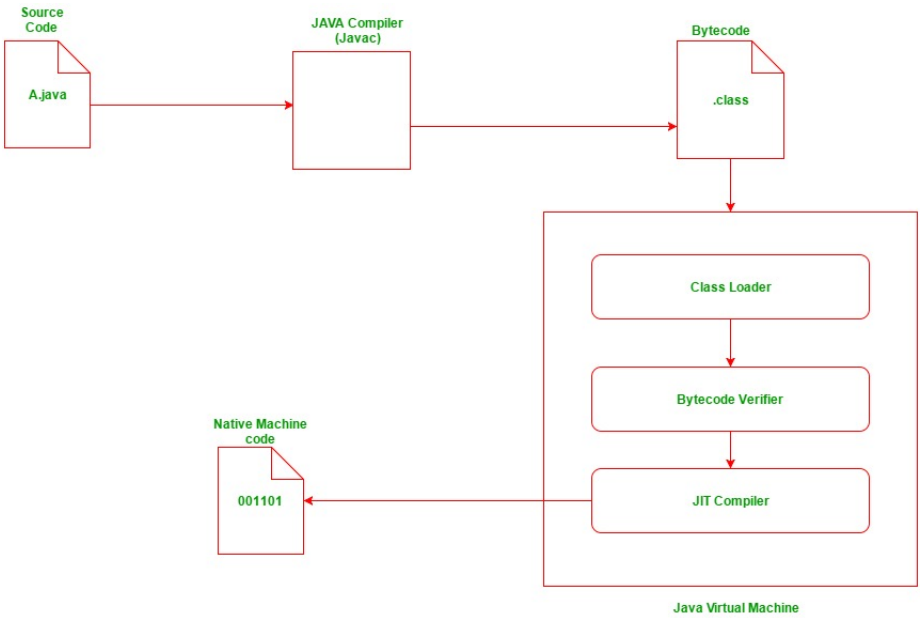


Figura 4.2: Esquema de ejecución de un programa en Java

4.2– Semanticas

Se repasará de nuevo la semántica del lenguaje IMP aportando nuevas formas de definirla (Small-steps o Big-Steps) y se estudiará la equivalencia entre ellas.

Las semánticas *Small-Steps* se definen como un método para evaluar expresiones paso a paso. De manera formal, se confirma que la semántica *small-step* para una expresión E es una relación:

$$E \rightarrow : E \times E$$

Figura 4.3: Función de reducción

Esta relación se conoce como función de reducción. Este tipo de semántica describe la evolución de una expresión de manera detallada. Es capaz de describir incluso programas no terminativos (que no garantizan su reducción en un valor) como una cadena finita de transiciones $e0 \rightarrow e1 \rightarrow e2 \rightarrow etc....$ Un programa terminativo es uno tal que $e0 \rightarrow e1 \rightarrow e2 \rightarrow etc... \rightarrow v$ y termina en un valor v tal que: $\forall e' \in E, v \nrightarrow e'$.

En el otro extremo del espectro semántico tenemos las semánticas denotacionales *(denotational semantics)*. Estas asignan significado a cada expresión. Es una función de expresiones a su concretización: $[\![\cdot]\!] : E \rightarrow D$ (D es llamado el dominio). El espacio que compone estas concretizaciones (o denotaciones) puede no tener relación ninguna con el espacio sintáctico. Por ejemplo E podría ser un conjunto de expresiones que se evaluan como un número y D podría ser un conjunto de números como $\mathbb{N}$ o $\mathbb{R}$.

Las semánticas *big-steps* están más o menos en medio de estos dos tipos. Una definición semántica *big-step* sobre una expresión E y un conjunto de valores V es una relación $\Downarrow : E \times V$. Esta relaciona una expresión a su valor (o posibles múltiples valores si el lenguaje de la expresión no es determinista). A veces, el valor especial $\perp$ se usa para expresiones no terminativas.

Operacionalmente hablando, las semánticas *small-steps* se corresponden a definir cuidadosamente cada operación realizada por un intérprete o compilador para un lenguaje. Las semánticas *big-steps* solo definen el resultado final. Las semánticas denotacionales reflejan la interpretación matemática, que puede o no, tener que ver con lo que pasa en un ordenador (como se computa).

4.2.1. Big-Steps

Se redefine la semántica *big step* para la evaluación de comandos en IMP:

```
Inductive ceval : state -> com -> state -> Prop :=
  | E_Skip : forall st,
      SKIP / st ==> st
  | E_Ass  : forall st a1 n l,
      aeval st a1 = n ->
      (l ::= a1) / st ==> (update st l n)
  | E_Seq : forall c1 c2 st st' st'',
      c1 / st  ==> st' ->
      c2 / st' ==> st'' ->
      (c1 ; c2) / st ==> st''
  | E_IfTrue : forall st st' b1 c1 c2,
      beval st b1 = true ->
      c1 / st ==> st' ->
      (IFB b1 THEN c1 ELSE c2 FI) / st ==> st'
  | E_IfFalse : forall st st' b1 c1 c2,
      beval st b1 = false ->
      c2 / st ==> st' ->
      (IFB b1 THEN c1 ELSE c2 FI) / st ==> st'
  | E_WhileEnd : forall b1 st c1,
      beval st b1 = false ->
      (WHILE b1 DO c1 END) / st ==> st
  | E_WhileLoop : forall st st' st'' b1 c1,
      beval st b1 = true ->
      c1 / st ==> st' ->
      (WHILE b1 DO c1 END) / st' ==> st'' ->
      (WHILE b1 DO c1 END) / st ==> st''

where "c1 '/' st '==>' st'" := (ceval st c1 st').
```

4.2.2. Small-Steps

Este tipo de semántica se representa como una reducción paso a paso: c / st → c'/ st'. Lo que significa que el comando c, ejecutado en el estado inicial st, realiza una transformación elemental. st

es el estado resultado tras ejecutar el comando c, el cual captura toda la computación (o reducciones) restante.

Las definiciones aritméticas y booleanas junto con sus evaluaciones en IMP que se encuentran en su correspondiente apartado teórico pueden verse como definiciones *small-steps* donde además se explicita también su ejecución paso a paso. Se parte de esta semántica para seguir tratando las evaluaciones aritméticas y booleanas como una sola definición *big step*. Como representan las siguientes reglas de inferencia (u operaciones semánticas) *CS_Ass, CS_IfTrue y CS_IfFalse*.

```
Inductive cstep : (com * state) -> (com * state) -> Prop :=
  | CS_Ass : forall st i a n,
      aeval st a = n ->
      (i ::= a) / st --> SKIP / (update st i n)
  | CS_SeqStep : forall st c1 c1' st' c2,
      c1 / st --> c1' / st' ->
      (c1 ; c2) / st --> (c1' ; c2) / st'
  | CS_SeqFinish : forall st c2,
      (SKIP ; c2) / st --> c2 / st
  | CS_IfTrue : forall st b c1 c2,
      beval st b = true ->
      IFB b THEN c1 ELSE c2 FI / st --> c1 / st
  | CS_IfFalse : forall st b c1 c2,
      beval st b = false ->
      IFB b THEN c1 ELSE c2 FI / st --> c2 / st
  | CS_While : forall st b c1,
      (WHILE b DO c1 END) / st
       --> (IFB b THEN (c1; (WHILE b DO c1 END)) ELSE SKIP FI) / st

where " c '/' st '-->' c' '/' st' " := (cstep (c,st) (c',st')).

Lemma cstep_deterministic:
  forall cs cs1, cstep cs cs1 -> forall cs2, cstep cs cs2 -> cs1 = cs2.
Proof.
  induction 1; intros cs2 CSTEP; inversion CSTEP; subst.
  auto.
  generalize (IHcstep _ H4). congruence.
  inversion H.
  inversion H3.
  auto.
  auto.
  congruence.
  congruence.
  auto.
  auto.
Qed.
```

La evaluación no atómica de expresiones resulta bastante compleja cuando se ejecutan de forma paralela varios comandos. En un sentido puramente secuencial, es equivalente (y mucho más simple) evaluar expresiones mediante pasos atómicos, dado que sus evaluaciones siempre son terminativas. Resumiendo lo anterior:

- Tres reglas computacionales: CS_Ass, CS_IfTrue y CS_IfFalse que son las encargas de realizar el groso de la reducción.
- Dos reglas de relocalización: CS_SeqFinish y CS_While que se encargan de cambiar el foco de reducción en otras evaluaciones.
- Una regla de contexto: CS_SeqStep que nos permite reducir una expresión de manera interna.

Siguiendo el estilo *small step*, las semánticas de un comando *c* en un estado *st* vienen dadas por la secuencia de reducciones necesarias empezando desde c, st. Esta secuencia puede ser:

- Una secuencia finita: cero, uno o varias reducciones
- Una secuencia infinita: Infinitas reducciones.

Estas secuencias están implementadas usando los operadores genéricos *star* y *infseq* aplicados a relaciones binarias, los cuales están definidos en el módulo *Sequences* junto con multitud de lemas útiles.

```
    Notation " c '/' st '-->*' c' '/' st' " := (star cstep (c, st) (c', st'))
(at level 40, st at level 39, c' at level 39).

Notation " c '/' st '-->' '' " := (infseq cstep (c, st))
(at level 40, st at level 39).

Definition terminates (c: com) (st: state) (st': state) : Prop :=
  c / st -->* SKIP / st'.

Definition diverges (c: com) (st: state) : Prop :=
  c / st --> .

Definition goes_wrong (c: com) (st: state) : Prop :=
  exists c', exists st',
  c / st --> c' / st' /\ irred cstep (c', st') /\ c' <> SKIP.
```

Generalización de la regla de CS_SeqStep. Demostrando que en una reducción concreta de una secuencia de reducciones se puede dar otra secuencia de reducciones interna:

```
    Lemma star_CS_SeqStep:
  forall c2 st c st' c',
  c / st -->* c' / st' ->
  (c;c2) / st -->* (c';c2) / st'.
    Proof.
      intros. dependent induction H.
      apply star_refl.
      destruct b as [st1 c1].
      eapply star_step. apply CS_SeqStep. eauto. auto.
    Qed.
```

Este lema se puede utilizar ahora para reducir una expresión reductible dentro de una secuencia de reducciones, lo cual se utilizará para demostrar la equivalencia entre:

- La terminación de una expresión de acuerdo a las semánticas *big-steps*.
- La existencia de una secuencia finita de reducciones que resultan en SKIP de acuerdo con las semántica *small-steps*

Se formalizan las implicaciones deducidas de transformar de la semántica *big-step* a la *small-step*, lo cual se puede conseguir de manera directa utilizando inducción sobre la evaluación de la expresiones en semántica *big-step*

```
  Theorem ceval_to_csteps:
forall c st st',
c / st ==> st' ->
c / st -->* SKIP / st'.
  Proof.
    induction 1.
  Case "SKIP".
    apply star_refl.
  Case ":=".
    apply star_one. apply CS_Ass. auto.
  Case "seq".
    eapply star_trans. apply star_CS_SeqStep. apply IHceval1.
    eapply star_step. apply CS_SeqFinish. auto.
  Case "if true".
    eapply star_step. apply CS_IfTrue. auto. auto.
  Case "if false".
    eapply star_step. apply CS_IfFalse. auto. auto.
  Case "while stop".
    eapply star_step. apply CS_While.
    apply star_one. apply CS_IfFalse. auto.
  Case "while true".
    eapply star_step. apply CS_While.
    eapply star_step. apply CS_IfTrue. auto.
    eapply star_trans. apply star_CS_SeqStep. apply IHceval1.
    eapply star_step. apply CS_SeqFinish. auto.
  Qed.
```

Las implicaciones inversas de pasar de *small-step* a *big-step* son más sutiles. La idea principal es mostrar que al aplicar una reducción y evaluarlo como una evaluación *big-step* que resulta a un estado final puede resumirse como una única evaluación *big-step* que lleve directamente al estado final.

```
  Lemma cstep_append_ceval:
forall c1 st1 c2 st2,
c1 / st1 --> c2 / st2 ->
forall st3,
c2 / st2 ==> st3 ->
c1 / st1 ==> st3.
  Proof.
    intros until st2. intro STEP. dependent induction STEP; intros.
  Case "CS_Ass".
    inversion H0; subst. apply E_Ass. auto.
  Case "CS_SeqStep".
    inversion H; subst. apply E_Seq with st'. eauto. auto.
  Case "CS_SeqFinish".
    apply E_Seq with st2. apply E_Skip. auto.
  Case "CS_IfTrue".
    apply E_IfTrue; auto.
  Case "CS_IfFalse".
```

```
    apply E_IfFalse; auto.
  Case "CS_While".
    inversion H; subst.
    SCase "while loop".
      inversion H6; subst.
      apply E_WhileLoop with st'; auto.
    SCase "while end".
      inversion H6; subst.
      apply E_WhileEnd; auto.
  Qed.
```

Como consecuencia de lo demostrado anteriormente, un término o expresión que se reduce y termina en SKIP es equivalente a evaluarlo con semántica *big-step*, lo que lleva en ambos casos al mismo estado final.

```
  Theorem csteps_to_ceval:
forall st c st',
c / st -->* SKIP / st' ->
c / st ==> st'.
  Proof.
    intros. dependent induction H.
    apply E_Skip.
    destruct b as [st1 c1]. apply cstep_append_ceval with st1 c1; auto.
  Qed.
```

4.2.3. Semánticas big-steps coinductivas para expresiones divergentes

Se define el predicado c / st $\rightarrow \infty$ que representa las evaluaciones divergentes. Los únicos comandos que pueden divergir son:

- c1 ; c2. Siempre que c1 diverja, o c1 termine y c2 diverja.
- IFB b THEN c1 ELSE c2 FI. Siempre que b sea *true* y c1 diverja. O b sea *false* y b2 diverja.
- WHILE b DO c END. Siempre que c diverja en la primera iteración o el bucle diverja en futuras iteraciones.

Se aporta ahora una definición para la evaluación de comandos que divergen:

```
  Reserved Notation " c '/' st '==>' '' " (at level 40, st at level 39).

CoInductive cevalinf: com -> state -> Prop :=
  | Einf_Seq_1: forall c1 c2 st,
      c1 / st ==>  ->
      (c1; c2) / st ==>
  | Einf_Seq_2: forall c1 c2 st1 st2,
      c1 / st1 ==> st2 -> c2 / st2 ==>  ->
      (c1; c2) / st1 ==>
  | Einf_IfTrue: forall b c1 c2 st,
      beval st b = true ->
      c1 / st ==>  ->
      (IFB b THEN c1 ELSE c2 FI) / st ==>
  | Einf_IfFalse: forall b c1 c2 st,
      beval st b = false ->
```

```
      c2 / st ==>  ->
      (IFB b THEN c1 ELSE c2 FI) / st ==>
  | Einf_WhileBody: forall b c1 st,
      beval st b = true ->
      c1 / st ==>  ->
      (WHILE b DO c1 END) / st ==>
  | Einf_WhileLoop: forall b c1 st st',
      beval st b = true ->
      c1 / st ==> st' -> (WHILE b DO c1 END) / st' ==>  ->
      (WHILE b DO c1 END) / st ==>

where " c '/' st '==>' '' " := (cevalinf c st).
```

El predicado c / st $\rightarrow \infty$ se describe como coinductivo debido a que interesa estudiar ambas derivaciones: las finitas y las infinitas. Si se describiera como inductivo, solo las finitas serían aceptadas y el predicado sería siempre falso, ya que no hay axiomas para empezar las que derivan en infinito.

Para demostrar que c / st $\rightarrow \infty$ se cumple para algún c y st, se deben usar demostraciones coinductivas. Estas demostraciones demuestran la existencia de la derivación asumiendo el resultado, luego se demuestra los pasos iniciales de la derivación usando uno o varios constructores del predicado coinductivo, y luego usa las hipótesis coinductivas previamente asumidas como argumento para alguno de estos constructores.

Como ejemplo, se demuestra que WHILE BTrue DO SKIP END siempre diverje:

```
    Remark while_true_skip_diverges:
  forall st, (WHILE BTrue DO SKIP END) / st ==> .
Proof.
  cofix COINDHYP.
  intros.
  apply Einf_WhileLoop with st.
  auto.
  apply E_Skip.
  apply COINDHYP.
Qed.
```

Para comprobar que la noción de coinductividad es correcta, se demuestra que es equivalente a la existencia de una secuencia infinita de reducciones. Se empieza con las implicaciones de la coinductividad *big-step* sobre las reducciones infinitas. El axioma principal es que un comando que diverge siguiendo la definición *big-step* siempre se puede reducir, e incluso su resultado diverge también.

```
Lemma cevalinf_can_progress:
  forall c st,
  c / st ==>  -> exists c', exists st', c / st --> c' / st' /\ c' / st' ==> .
Proof.
      induction c; intros st CINF; inversion CINF; subst.
    Case "Seq1".
      destruct (IHc1 _ H2) as [c1' [st' [A B]]].
      exists (c1';c2); exists st'; split.
      apply CS_SeqStep; auto.
```

```
      eapply Einf_Seq_1; eauto.
    Case "Seq2".
      assert (ST: star cstep (c1, st) (SKIP, st2)). apply ceval_to_csteps; auto.
      inversion ST; subst.
      SCase "Seq2 skip".
        exists c2; exists st2; split.
        apply CS_SeqFinish.
        auto.
      SCase "Seq2 notskip".
        destruct b as [c1' st'].
        exists (c1';c2); exists st'; split.
        apply CS_SeqStep; auto.
        apply Einf_Seq_2 with st2. apply csteps_to_ceval; auto. auto.
    Case "IfTrue".
      exists c1; exists st; split.
      apply CS_IfTrue; auto.
      auto.
    Case "IfFalse".
      exists c2; exists st; split.
      apply CS_IfFalse; auto.
      auto.
    Case "WhileBody".
      exists (IFB b THEN (c; (WHILE b DO c END)) ELSE SKIP FI); exists st; split.
      apply CS_While.
      apply Einf_IfTrue; auto. apply Einf_Seq_1; auto.
    Case "WhileLoop".
      exists (IFB b THEN (c; (WHILE b DO c END)) ELSE SKIP FI); exists st; split.
      apply CS_While.
      apply Einf_IfTrue; auto. apply Einf_Seq_2 with st'; auto.
Qed.
```

Iterando el lema anterior de forma infinita, se puede construir una secuencia infinita de reducciones:

```
    Lemma cevalinf_diverges:
  forall c st,
  c / st ==>  -> c / st --> .
    Proof.
      cofix COINDHYP; intros.
      destruct (cevalinf_can_progress c st H) as [c' [st' [A B]]].
      apply infseq_step with (c', st'). auto. apply COINDHYP. auto.
    Qed.
```

Para la implicación inversa: de una secuencia infinita de reducciones a la divergencia coinductiva *big-step*. Se utiliza una serie de lemas de inversión para descomponer la secuencia de reducciones infinitas c / st $\rightarrow \infty$ en subsecuencias:

```
    Lemma diverges_skip_impossible:
  forall st, ~(SKIP / st --> ).
    Proof.
      intros; red; intros. inversion H; subst. inversion H0.
    Qed.
```

```
Lemma diverges_assign_impossible:
  forall v a st, ~((v ::= a) / st --> ).
    Proof.
      intros; red; intros. inversion H; subst. inversion H0; subst.
      inversion H1; subst. inversion H2.
    Qed.

Ltac inv H := inversion H; subst; clear H.
```

Los siguientes lemas de inversión para secuencias usan el hecho de que una secuencia de reducción que es infinita o finita resulta en un estado irreducible. Esto es intuitivo, pero es imposible de demostrar mediante Coq debido a que solo admite demostraciones constructivas. Se necesita de la lógica clásica para obtener esta demostración:

```
    Lemma diverges_seq_inversion:
  forall st c1 c2,
  (c1;c2) / st -->  ->
  c1 / st -->  \/ (exists st', c1 / st -->* SKIP / st' /\ c2 / st' --> ).
Proof.
  intros.
  destruct (infseq_or_finseq cstep (c1, st)) as [DIV1 | [[c' st'] [RED1 IRRED1]]].
Case "c1 diverges".
  auto.
Case "c1 terminates".
  assert ((c'; c2) / st' --> ).
    eapply infseq_star_inv; eauto.
    intros; eapply cstep_deterministic; eauto.
    apply star_CS_SeqStep; auto.
  inv H0.
  inv H1.
  elim (IRRED1 _ H6).
  right; exists st'; auto.
Qed.

Lemma diverges_ifthenelse_inversion:
  forall b c1 c2 st,
  (IFB b THEN c1 ELSE c2 FI) / st -->  ->
  (beval st b = true /\ c1 / st --> ) \/ (beval st b = false /\ c2 / st --> ).
Proof.
  intros. inv H. inv H0; auto.
Qed.

Lemma diverges_while_inversion:
  forall b c st,
  (WHILE b DO c END) / st -->  ->
  beval st b = true /\
  (c / st -->  \/ (exists st', c / st -->* SKIP / st' /\ (WHILE b DO c END) / st' --> )).
Proof.
  intros. inv H. inv H0. inv H1. inv H.
Case "b is true".
  split. auto. apply diverges_seq_inversion. auto.
Case "b is false".
```

```
  elim (diverges_skip_impossible _ H0).
Qed.

Lemma diverges_to_cevalinf:
forall c st, c / st -->  -> c / st ==> .
Proof.
cofix COINDHYP; intros.
destruct c.
Case "skip".
elim (diverges_skip_impossible _ H).
Case ":=".
elim (diverges_assign_impossible _ _ _ H).
Case "seq".
destruct (diverges_seq_inversion _ _ _ H) as [DIV1 | [st' [TERM1 DIV2]]].
apply Einf_Seq_1. apply COINDHYP; auto.
apply Einf_Seq_2 with st'. apply csteps_to_ceval; auto. apply COINDHYP; auto.
Case "if".
destruct (diverges_ifthenelse_inversion _ _ _ _ H) as [[TRUE DIV1] | [FALSE DIV2]].
apply Einf_IfTrue. auto. apply COINDHYP; auto.
apply Einf_IfFalse. auto. apply COINDHYP; auto.
Case "while".
destruct (diverges_while_inversion _ _ _ H) as [TRUE [DIV1 | [st' [TERM1 DIV2]]]].
apply Einf_WhileBody. auto. apply COINDHYP; auto.
apply Einf_WhileLoop with st'. auto. apply csteps_to_ceval; auto. apply COINDHYP; auto.
Qed.
```

4.3– Compilador

Una vez definidas las secuencias de reducciones semánticas para las expresiones del lenguaje, se pasa a comprobar que la equivalencia de esta semántica se mantiene incluso al cambiar el lenguaje. Para ello se construirá un compilador que traducirá desde el lenguaje IMP a una máquina virtual inspirada en un pequeño conjunto de instrucciones de la máquina virtual de Java.

La máquina virtual estará formada por tres componentes básicos que irán evolucionando y actualizándose a cada iteración del estado:

- Un puntero de instrucciones *(Program counter)* que irá apuntado a la posición actual de la instrucción en ejecución.
- Un estado *(State)*, que se encargará de almacenar los valores de las variables.
- Una pila de evaluación *(Stack)* que se encargará de almacenar y actualizar los valores de las variables para realizar las distintas operaciones.

Todo esto estará sujeto al código c que será una lista fija de instrucciones. Sobre la cual la máquina virtual establecerá la posición de la instrucción *(program counter)* almacenará la asignación de las variables que indique el código *(state)* y actualizará dichos valores en función de la instrucción en la pila *(stack)*.

4.3.1. Instrucciones

El conjunto de instrucciones de la máquina virtual será el siguiente:

- Una instrucción para introducir un valor.
- Una instrucción para introducir una variable.
    - Una instrucción para asignar una variable.
- Tres instrucciones para operaciones aritméticas:
    - Una instrucción para sumar dos valores.
    - Una instrucción para restar dos valores.
    - Una instrucción para multiplicar dos valores.
- cuatro instrucciones para operaciones booleanas:
    - Una instrucción para evaluar la igualdad.
    - Una instrucción para evaluar la desigualdad.
    - Una instrucción para evaluar menor o igual que.
    - Una instrucción para evaluar mayor que.
- Dos instrucciones para saltos de instrucción:
    - Una instrucción para saltar adelante.
    - Una instrucción para saltar atrás.
- Una instrucción de terminación.

Se define así el conjunto de instrucciones. También se define el código *(code)* como una lista de instrucciones, y la pila *(stack)* como una lista de números.

Se necesitará también de un método que indique la posición de la instrucción a compilar, o que indique la instrucción concreta. Este detalle depende totalmente del diseño escogido. Lo importante es que encapsule el concepto de puntero de programa (program counter) que se trata, en la mayoría de computadores, de un registro que apunta a la instrucción en ejecución.

4.3.2. Transiciones

La semántica de la máquina virtual se define de forma *small-step* como relaciones de transición entre los diferentes estados de la máquina. Un estado máquina se define como una tripleta formada por los valores del puntero de instrucciones, la pila y el estado *(program counter, stack, state)*

Esta relación de transición viene completamente sujeta a la instrucción en ejecución del código c. Existe solo una regla de transición para cada instrucción excepto para la instrucción de terminación.

Como es normal con las semánticas *small-steps*, se forman secuencias de transiciones entre estados máquina para definir el comportamiento del código. Se empieza siempre con el puntero de instrucciones *(program counter, pc)* pc = 0 y con la pila de evaluación vacía. El programa se da por terminado siempre que pc apunte a una instrucción *lhalt* y la pila de evaluación esté vacía. Si R es una relación binaria, *star* R es su clausura reflexiva transitiva. Esto es:

Se supone la relación R en un conjunto A. La clausura reflexiva transitiva de R es la relación más pequeña S aplicada a A que cumple:

1. $R \subseteq S$.
2. S es reflexiva.
3. S es transitiva.

Por lo que *star (transition C)* representa una secuencia de cero, uno o más transiciones de estados máquina.

```
  Definition mach_terminates (C: code) (s_init s_fin: state) :=
exists pc,
code_at C pc = Some Ihalt /\
star (transition C) (0, nil, s_init) (pc, nil, s_fin).
```

Derivado de esto, *infseq R* representa una sucesión infinita de R transiciones.

```
  Definition mach_diverges (C: code) (s_init: state) :=
infseq (transition C) (0, nil, s_init).

  Definition mach_goes_wrong (C: code) (s_init: state) :=
  exists pc, exists stk, exists s_fin,
  star (transition C) (0, nil, s_init) (pc, stk, s_fin)
  /\ irred (transition C) (pc, stk, s_fin)
  /\ (code_at C pc <> Some Ihalt \/ stk <> nil).
```

Esto concuerda con la definición *small-step* aportada en el capítulo de semántica sobre las secuencias de reducciones para los distintos estados.

4.3.3. Esquema de compilación

En cuanto a semántica esto es equivalente a la evaluación de las distintas expresiones. Sin embargo en este caso se habla de compilación debido a que se trata con código (que es la concretización de la expresiones semánticas).

4.3.4. Expresiones aritméticas

El código para una expresión aritmética a:

- Se ejecuta de manera secuencial.
- Introduce el valor de a al inicio de la pila.
- Mantiene el estado de la variable.

Esta definición se explicitará en notación polaca inversa *(Reverse Polish Notation, RPN)* típica de las estructuras tipo pila. Esta es un método algebraico alternativo para la introducción de datos: Primero se define los operandos y después el operador que los relaciona.

Algunos ejemplos para la compilación de expresiones:

```
Notation vx := (Id 0).
Notation vy := (Id 1).

Eval compute in (compile_aexp (APlus (AId vx) (ANum 1))).

Eval compute in (compile_aexp (AMult (AId vy) (APlus (AId vx) (ANum 1)))).
```

4.3.5. Expresiones booleanas

En el caso de las expresiones booleanas, el código para una expresión booleana b:

- Salta adelante *ofs* instrucciones cuando se evalúa como una condición booleana, *cond*, true.
- Ejecuta secuencialmente si b se evalúa como la negación de *cond*
- Deja la pila y el estado sin cambios.

Algunos ejemplos:

```
Eval compute in (compile_bexp (BEq (AId vx) (ANum 1)) true 42).

Eval compute in (compile_bexp (BAnd (BLe (ANum 1) (AId vx)) (BLe (AId vx) (ANum 10)))
                              false
                              42).

Eval compute in (compile_bexp (BNot (BAnd BTrue BFalse)) true 42).
```

4.3.6. Comandos

El código para un comando c:

- Actualiza el estado de la variable como dictamina c.
- Mantiene la pila.
- Termina en la siguiente instrucción siguiendo con el código generado.

Se entiende como comandos las distintas estructuras de control y orden para las instrucciones de un programa. Estas serán las que determinen el orden de ejecución de las instrucciones y evalúen las condiciones lógicas necesarias para hacer funcionar las distintas estructuras de control típicas de cualquier lenguaje de programación.

4.4– Deadcode

En este subcapítulo se expondrá el concepto de análisis de la vida útil de las variables *(live-variable analysis, liveness analysis)* y su uso para optimizar el compilador en lo que se denomina eliminación del código muerto *(deadcode elimination)*.

4.4.1. Conjuntos de variables

En primer lugar, cabe destacar que el análisis estático que se necesita; el *liveness analysis*, opera sobre conjuntos de variables. La librería estándar de Coq provee de una más que suficiente implementación para operar con conjuntos finitos. Antes de usarse, se necesita aportar un módulo que permita definir un criterio de igualdad y orden para los tipos de los identificadores (en este caso el valor de las variables).

En este módulo se explicitarán el conjunto de demostraciones necesarios para garantizar propiedades necesarias para comparar los identificadores de las variables. Estableciendo un concepto de igualdad, comparación y identidad para cada variable.

Así mismo se necesitará de una función de comparación basada en estas propiedades. Esta función servirá para definir las operaciones necesarias para comparar variables.

Por último, se instancian los módulos de la librería estándar de Coq para trabajar con estos conjuntos finitos de variables indicando como orden a seguir el desarrollado en el módulo anterior.

Estos módulos implementan conjuntos finitos usando árboles AVL siguiendo la implementación de la librería estándar de OCaml.

4.4.2. *Liveness analysis*

Para definir el algoritmo de vivacidad de las variables será necesario establecer una evaluación de las variables en función del conjunto al que pertenecen.

Estas evaluaciones tienen que soportar los distintos tipos de expresiones que pueden encontrarse en el compilador, además de los comandos propios de las instrucciones del código. Estas evaluaciones permiten agrupar todas las asignaciones de variables en un conjunto. Que más tarde se estudiará para ver que variables interesa mantener o pueden ser consideradas muertas.

Los distintos tipos *VS.empty, VS.singleton y VS.union* hacen referencia a los tipos de conjunto vacío, conjunto unitario (un único valor) y conjunto unión, respectivamente. Todos ellos pertenecientes a la librería de Coq.

Se define L como el conjunto de variables vivas después del comando c. El resultado de live c L es el conjunto de variables vivas antes de c.

Se entiende que una variable está viva cuando esta ha sido leída o utilizada por alguna otra variable o estructura de control. Se entiende que está muerta cuando no ocurre esto. El análisis de vivacidad *(liveness analysis)* es un algoritmo que se ejecuta desde el final del código hasta el principio del mismo. Analizando cada bloque de código y cada bifurcación del mismo asignando a cada variable a un posible conjunto: el de las variables vivas o muertas.

En función de esta asignación se establece qué variables pueden ser desechadas ya que al estar muertas no intervienen en el comportamiento del programa.

4.4.3. Eliminación del código muerto

Para realizar la correcta eliminación de las variables pertenecientes al conjunto de variables muertas se define una función que se encarga de cambiar las asignaciones de dichas variables en el código en comandos SKIP, los cuales son inocuos para el comportamiento del programa.

Ejemplo de optimización:

```
    Notation va := (Id 0).
    Notation vb := (Id 1).
    Notation vq := (Id 2).
    Notation vr := (Id 3).

Definition prog :=
  ( vr ::= AId va;
    vq ::= ANum 0;
    WHILE BLe (AId vb) (AId vr) DO
      vr ::= AMinus (AId vr) (AId vb);
      vq ::= APlus (AId vq) (ANum 1)
    END ).

Eval vm_compute in (dce prog (VS.singleton vr)).
```

4.5– Regalloc

Se implementará ahora una optimización basada en la asignación de registros *(register allocation)* enfocada como un cambio de nombre de las variables de un programa en IMP manteniendo la semántica como precepto fundamental.

En computación, la asignación de registros *(register allocation)* es el proceso de asignar variables y resultados de expresiones a un número reducido de registros de procesamiento *(processor registers)*. Estos registros de procesamiento son registros que tienen un acceso rápido al procesador de un ordenador, por lo tanto son más eficientes y rápidos que los registros normales.

La asignación de registros puede aplicarse tanto a bloques básicos de memoria como a una función o proceso. El objetivo de esta optimización es asignar la mayor cantidad de variables al mayor número de registros del procesador para que estén almacenadas en una situación privilegiada para facilitar su lectura, escritura y computación. Las variables que no pueden ser asignadas al procesador pasarían a estar almacenadas en la memoria RAM, la cual es significativamente menos eficaz que estos registros.

4.5.1. Reasignación de las variables

Para simplificar la situación, se entenderá que una variable ha sido asignada a un registro del procesador cuando esta sea renombrada por otra variable. Primero se necesita reconocer aquellas expresiones que son solo variables. Por lo tanto se implementa una función que devuelve el id de la variable siempre que esta expresión se evalúe como tal

Se necesitará también una nueva evaluación para las expresiones aritméticas y booleanas que permita cambiar la asignación de la variable. Además de un método auxiliar que se encarge de renombrar la variable respetando todas sus apariciones futuras.

4.5.2. Transformación de los comandos

Por lo tanto, utilizando estas nuevas funciones se puede implementar una función de optimización que transforme el código de un programa en IMP en función de los comandos a ejecutar. Por mejorar la optimización, aquellas variables que sean asignaciones de variables de la forma $x = y$ donde $fx = fy$ (variables fútiles) serán convertidas en comandos SKIP

Esta función necesitará tener en cuenta todos los posibles estados de los comandos del programa a compilar.

4.5.3. Condiciones para la reasignación de variables

A continuación se recogen el conjunto de propiedades necesarias para realizar una selección óptima de las variables que serán renombradas. Estas también son necesarias para garantizar la preservación semántica de los programas resultado. Todas estas propiedades o condiciones son acuerdos o asunciones que se necesitan corroborar, sirven como una heurística para decidir que conjunto de variables serán las reasignadas.

Primero se define la propiedad de la monotonicidad de la implicación, la cual afirma que las hipótesis de cualquier hecho derivado pueden ser extendidas libremente con supuestos adicionales. Esto sirve como una regla de debilitamiento para el conjunto de restricciones impuestas.

```
    Lemma agree_mon:
  forall L L' s1 s2,
  agree L' s1 s2 -> VS.Subset L L' -> agree L s1 s2.
    Proof.
      unfold VS.Subset, agree; intros. auto.
    Qed.

Lemma agree_extensional:
  forall L s1 s2 s3,
  agree L s1 s2 -> (forall x, s2 x = s3 x) -> agree L s1 s3.
    Proof.
      unfold agree; intros. transitivity (s2 (f x)); auto.
    Qed.
```

Se define ahora la condición sobre las variables libres de una expresión que implica que tanto la expresión como sus variables renombradas se evalúan de igual manera en ambos estados:

```
Lemma aeval_agree:
forall L s1 s2, agree L s1 s2 ->
forall a, VS.Subset (fv_aexp a) L ->
aeval s1 a = aeval s2 (rename_aexp a).
Proof.
  induction a; simpl; intros.
  auto.
  apply H. generalize H0; fsetdec.
  f_equal. apply IHa1. generalize H0; fsetdec. apply IHa2. generalize H0; fsetdec.
  f_equal. apply IHa1. generalize H0; fsetdec. apply IHa2. generalize H0; fsetdec.
  f_equal. apply IHa1. generalize H0; fsetdec. apply IHa2. generalize H0; fsetdec.
Qed.

Lemma beval_agree:
forall L s1 s2, agree L s1 s2 ->
forall b, VS.Subset (fv_bexp b) L ->
beval s1 b = beval s2 (rename_bexp b).
Proof.
  induction b; simpl; intros.
  auto.
  auto.
  repeat rewrite (aeval_agree L s1 s2); auto; generalize H0; fsetdec.
  repeat rewrite (aeval_agree L s1 s2); auto; generalize H0; fsetdec.
  f_equal; auto.
  f_equal. apply IHb1. generalize H0; fsetdec. apply IHb2. generalize H0; fsetdec.
Qed.
```

Dada esta condición se pasa a formalizar y corroborar que se cumple en los distintos casos de optimización:

- La condición se preserva entre asignaciones unilaterales y variables muertas

```
Lemma agree_update_dead:
  forall s1 s2 L x v,
  agree L s1 s2 -> ~VS.In x L ->
```

```
  agree L (update s1 x v) s2.
Proof.
  intros; red; intros. unfold update. remember (beq_id x x0). destruct b.
  assert (x = x0). apply beq_id_eq. auto. subst x0. contradiction.
  apply H; auto.
Qed.
```

- La condición se preserva dada una asignación simultánea de una variable x y su reasignación f x, dado que ninguna de las otras variables vivas z han sido ya reasignada a f x

```
Lemma agree_update_live:
  forall s1 s2 L x v,
  agree (VS.remove x L) s1 s2 ->
  (forall z, VS.In z L -> z <> x -> f z <> f x) ->
  agree L (update s1 x v) (update s2 (f x) v).
Proof.
  intros; red; intros. unfold update. remember (beq_id x x0). destruct b.
Case "x = x0".
  assert (x = x0). apply beq_id_eq. auto. subst x0.
  rewrite <- beq_id_refl. auto.
Case "x <> x0".
  assert (x <> x0). apply beq_id_false_not_eq; auto.
  assert (f x0 <> f x). apply H0; auto.
  rewrite not_eq_beq_id_false.
  apply H. apply VS.remove_2; auto.
  auto.
Qed.
```

4.5.4. Función de reasignado de variables

Juntando todas las comprobaciones anteriores sobre la condición inicial se obtiene un criterio para decidir las posibles variables candidatas a ser renombradas mediante la función f. Esta es una función que devuelve un booleano, no un predicado. Por lo tanto puede usarse para asegurar el correcto renombrado independientemente de la heurística escogida.

Este criterio queda definido como:

```
Fixpoint correct_allocation (c: com) (L: VS.t) : bool :=
match c with
| SKIP =>
  true
| x ::= a =>
  if VS.mem x L then
    match expr_is_var a with
    | Some y =>
        VS.for_all (fun z => beq_id z x || beq_id z y || negb (beq_id (f z) (f x))) L
    | None =>
        VS.for_all (fun z => beq_id z x || negb (beq_id (f z) (f x))) L
    end
  else true
| (c1; c2) =>
```

```
  correct_allocation c1 (live c2 L) && correct_allocation c2 L
| IFB b THEN c1 ELSE c2 FI =>
  correct_allocation c1 L && correct_allocation c2 L
| WHILE b DO c END =>
  correct_allocation c (live (WHILE b DO c END) L)
end.
```

CAPÍTULO 5

Análisis de requisitos, diseño e implementación

5.1– Compilador

Se desarrollarán ahora los requisitos necesarios para el compilador en función de sus componentes individuales, justificando las decisiones de diseño y por último, aportando su implementación.

El compilador a desarrollar tendrá que responder a las siguientes necesidades o requisitos:

- Tiene que ser un sistema a prueba de fallos.
- Tiene que ser capaz de garantizar la preservación de la semántica del lenguaje.
- Tiene que ser un sistema determinista.
- Capacidad para evaluar expresiones aritméticas.
- Capacidad para evaluar expresiones booleanas.
- Capacidad para evaluar comandos propios de un lenguaje de programación.
- Una vez construido, el sistema debe ser optimizado en cuanto a rendimiento y almacenamiento.

El sistema tiene que ser a prueba de fallos, o mejor dicho, que el sistema se comporte de manera correcta, Esto quiere decir que se comporte de la manera en la que está especificado que lo haga. Para conseguir esto, se ha decidido formalizar el sistema describiendo sus comportamientos, estados y elementos de manera matemática y, posteriormente, se han acotado y demostrado sus posibles estados mediante la lógica.

Para garantizar la preservación de la semántica se han construido un conjunto de reglas semánticas para poder deducir el significado de las expresiones construidas. Posteriormente, con la ayuda de un asistente de demostraciones, y junto con estas reglas de inferencia, se demostrará que las expresiones resultado de la traducción del compilador son equivalentes semánticamente.

Para que el sistema sea determinista se ha utilizado las ventajas de la programación funcional y la especificación formal para poder crear un sistema seguro que determine cada uno de los posibles estados del espacio entero de posibilidades. Esto asegura que el compilador se comporta de manera constante ante la misma entrada de datos (en este caso, al traducir las instrucciones de un programa). Y que los estados futuros son consecuencia de estados previos.

5.1.1. Instrucciones

Requisitos

La máquina virtual necesitará de tres elementos básicos: una pila, un conjunto de instrucciones, y el código del programa.

La pila necesitará ser capaz de almacenar un conjunto de valores numéricos y necesitará también la capacidad de operar y manipular dichos valores. Para ello se crearán un conjunto de instrucciones que lo permitan.

Se necesita de un lugar donde almacenar el estado de las variables tratadas por la pila.

También se necesitará una forma de conocer la línea de código IMP a compilar, por lo que es necesario definir una función que lo proporcione.

El conjunto de instrucciones para la máquina virtual tiene que responder a las siguientes necesidades:

- Capacidad para introducir valores numéricos.
- Capacidad para definir y asignar variables.
- Capacidad para realizar operaciones aritméticas con dos valores.
- Capacidad para realizar operaciones lógicas con dos valores.
- Capacidad para gestionar el salto de instrucciones.
- Capacidad de reconocer la terminación de un programa.

Diseño

Para responder al conjunto de requisitos anterior, el conjunto de instrucciones necesario dispondrá de las siguientes instrucciones:

- Iconst: Que permite introducir un valor numérico en la cima de la pila. Para ello necesitará el valor numérico a añadir.
- Ivar: Que permite definir una variable, para ello necesitará el ID de dicha variable.
- Isetvar: Que permite extraer el último valor de la pila y asignárselo a la variable cuyo ID se identifique con el parámetro de entrada.
- Iadd: Que permite sumar los dos últimos valores de la pila e introducir el resultado de vuelta.
- Isub: Que permite restar los dos últimos valores de la pila e introducir el resultado de vuelta.

- Imul: Que permite multiplicar los dos últimos valores de la pila e introducir el resultado de vuelta.
- Ibranch_forward: Que permite avanzar adelante tantas instrucciones como indique el valor de entrada ofs.
- Ibranch_backward: Que permite avanzar atrás tantas instrucciones como indique el valor de entrada ofs.
- Ibeq: Que permite comprobar la condición lógica de que los dos últimos valores de la pila son iguales y saltar ofs instrucciones adelante si se cumple.
- Ibne: Que permite comprobar la condición lógica de que los dos últimos valores de la pila son distintos y saltar ofs instrucciones adelante si se cumple.
- Ible: Que permite comprobar la condición lógica de que el primer valor es menor o igual que el segundo y saltar ofs instrucciones adelante si se cumple.
- Ibgt: Que permite comprobar la condición lógica de que el primer valor es mayor que el segundo y saltar ofs instrucciones adelante si se cumple.
- Ihalt: Que permite indicar que la ejecución ha terminado de manera exitosa.

En cuánto al diseño de la pila, se creará una lista vacía de valores naturales que almacenará los valores indicados por las instrucciones.

Para almacenar el estado de las variables de forma persistente, así como para consultar sus valores, se creará un mapa que emulará el comportamiento de una memoria. Esta decisión se toma buscando ejemplificando el escenario real con una simplificación.

Para el código del programa IMP, se representará como una lista cerrada de instrucciones que se ejecutan de manera secuencial según su posición. Esta decisión de diseño permite simplificar el escenario a un entorno más entendible y controlado.

Vinculado a esto, se ha decidido crear un método auxiliar que dada la lista de código IMP y una posición como número natural, devolverá un opcional de la instrucción que se encuentra en dicha posición. La elección de devolver un opcional sirve como seguridad contra tipos incorrectos a la hora de devolver un resultado.

Implementación

La implementación para el conjunto de instrucciones es la siguiente:

```
Inductive instruction: Type :=
| Iconst(n: nat)
| Ivar(x: id)
| Isetvar(x: id)
| Iadd
| Isub
| Imul
| Ibranch_forward(ofs: nat)
| Ibranch_backward(ofs: nat)
| Ibeq(ofs: nat)
| Ibne(ofs: nat)
```

```
| Ible(ofs: nat)
| Ibgt(ofs: nat)
| Ihalt.
```

Para la pila, el estado y el código a compilar, la implementación queda como:

```
Definition stack := list nat.
Definition state := total_map nat.
Definition code := list instruction.
```

Se define a continuación el método *code_at* el cual devolverá la instrucción i que se encuentra en la posición pc de la lista de instrucciones c. Siempre que esta exista:

```
Fixpoint code_at (C: code) (pc: nat) : option instruction :=
  match C, pc with
  | nil, _ => None
  | i :: C', O => Some i
  | i :: C', S pc' => code_at C' pc'
  end.
```

5.1.2. Transiciones

Requisitos

Las transiciones deben responder a los siguientes requisitos:

- Describir cada posible estado del compilador.
- Describir el estado de la pila en función del estado.
- Describir el estado de las variables en función del estado.
- Describir la posición del código a compilar en función del estado.
- Describir la operación a realizar en función de la instrucción.

También se necesita una manera de definir cada estado máquina de manera concreta y precisa.

Diseño

Para definir el conjunto de posibles estados en el que se puede encontrar un compilador se han definido dichas transiciones:

- trans_const: Encargada de actualizar el program counter + 1 y de añadir el valor en la cima de la pila.
- trans_var: Encargada de actualizar el program counter + 1 y de inicializar la variable en la cima de la pila.
- trans_setvar: Encargada de actualizar el program counter + 1 y actualizar el estado con el valor de la variable.
- trans_add: Encargada de actualizar el program counter + 1 y de añadir a la pila la suma de los dos valores tope de la pila.
- trans_sub: Encargada de actualizar el program counter + 1 y de añadir a la pila la resta de los dos valores tope de la pila.

- trans_mul: Encargada de actualizar el program counter + 1 y de añadir a la pila la multiplicación de los dos valores tope de la pila.
- trans_branch_forward: Encargada de actualizar el program counter + 1 más el valor ofs.
- trans_branch_backward: Encargada de actualizar el program counter + 1 menos el valor ofs.
- trans_beq: Encargada de actualizar el program counter + 1 más ofs siempre que se cumpla la condición de igualdad entre los valores de la instrucción.
- trans_bne: Encargada de actualizar el program counter + 1 más ofs siempre que se cumpla la condición de desigualdad entre los valores de la instrucción.
- trans_ble: Encargada de actualizar el program counter + 1 más ofs siempre que se cumpla la condición de que el primer valor de la instrucción es menor o igual que el segundo.
- trans_bgt: Encargada de actualizar el program counter + 1 más ofs siempre que se cumpla la condición de que el primer valor de la instrucción es mayor que el segundo.

Para el definir el estado máquina se decide utilizar una triple de valores que representen los tres valores más indicativos de un estado:(estado del program counter, pila, estado).

Implementación

Se formaliza la definición de un estado máquina mediante sus valores más representativos:

```
Definition machine_state := (nat * stack * state)%type.
```

Se formaliza la relación de transición para cada instrucción de la máquina virtual:

```
Inductive transition (C: code): machine_state -> machine_state -> Prop :=
  | trans_const: forall pc stk s n,
      code_at C pc = Some(Iconst n) ->
      transition C (pc, stk, s) (pc + 1, n :: stk, s)
  | trans_var: forall pc stk s x,
      code_at C pc = Some(Ivar x) ->
      transition C (pc, stk, s) (pc + 1, s x :: stk, s)
  | trans_setvar: forall pc stk s x n,
      code_at C pc = Some(Isetvar x) ->
      transition C (pc, n :: stk, s) (pc + 1, stk, update s x n)
  | trans_add: forall pc stk s n1 n2,
      code_at C pc = Some(Iadd) ->
      transition C (pc, n2 :: n1 :: stk, s) (pc + 1, (n1 + n2) :: stk, s)
  | trans_sub: forall pc stk s n1 n2,
      code_at C pc = Some(Isub) ->
      transition C (pc, n2 :: n1 :: stk, s) (pc + 1, (n1 - n2) :: stk, s)
  | trans_mul: forall pc stk s n1 n2,
      code_at C pc = Some(Imul) ->
      transition C (pc, n2 :: n1 :: stk, s) (pc + 1, (n1 * n2) :: stk, s)
  | trans_branch_forward: forall pc stk s ofs pc',
      code_at C pc = Some(Ibranch_forward ofs) ->
```

```
    pc' = pc + 1 + ofs ->
    transition C (pc, stk, s) (pc', stk, s)
| trans_branch_backward: forall pc stk s ofs pc',
    code_at C pc = Some(Ibranch_backward ofs) ->
    pc' = pc + 1 - ofs ->
    transition C (pc, stk, s) (pc', stk, s)
| trans_beq: forall pc stk s ofs n1 n2 pc',
    code_at C pc = Some(Ibeq ofs) ->
    pc' = (if beq_nat n1 n2 then pc + 1 + ofs else pc + 1) ->
    transition C (pc, n2 :: n1 :: stk, s) (pc', stk, s)
| trans_bne: forall pc stk s ofs n1 n2 pc',
    code_at C pc = Some(Ibne ofs) ->
    pc' = (if beq_nat n1 n2 then pc + 1 else pc + 1 + ofs) ->
    transition C (pc, n2 :: n1 :: stk, s) (pc', stk, s)
| trans_ble: forall pc stk s ofs n1 n2 pc',
    code_at C pc = Some(Ible ofs) ->
    pc' = (if ble_nat n1 n2 then pc + 1 + ofs else pc + 1) ->
    transition C (pc, n2 :: n1 :: stk, s) (pc', stk, s)
| trans_bgt: forall pc stk s ofs n1 n2 pc',
    code_at C pc = Some(Ibgt ofs) ->
    pc' = (if ble_nat n1 n2 then pc + 1 else pc + 1 + ofs) ->
    transition C (pc, n2 :: n1 :: stk, s) (pc', stk, s).
```

5.1.3. Evaluación aritmética

Requisitos

La evaluación aritmética de las expresiones en IMP al ser compiladas necesitan responder a:

- Mantener el orden del código en IMP.
- Capacidad de evaluar un valor.
- Capacidad de evaluar una variable.
- Capacidad de evaluar las distintas operaciones aritméticas.

Diseño

Para responder a las necesidades anteriores, la evaluación del código en IMP queda reflejada como:

- Se construirá como un tipo recursivo para respetar el orden secuencial de ejecución del código.
- ANum: Que se evalúa como un valor.
- AId: Que se evalúa como una asignación de variable.
- APlus: Que se evalúa como una suma de dos expresiones aritméticas.
- AMinus: Que se evalúa como una resta de dos expresiones aritméticas.
- AMult: Que se evalúa como una multiplicación de dos expresiones aritméticas.

Implementación

Por lo tanto, la compilación de las expresiones aritméticas queda definida como:

```
Fixpoint compile_aexp (a: aexp) : code :=
  match a with
  | ANum n => Iconst n :: nil
  | AId v => Ivar v :: nil
  | APlus a1 a2 => compile_aexp a1 ++ compile_aexp a2 ++ Iadd :: nil
  | AMinus a1 a2 => compile_aexp a1 ++ compile_aexp a2 ++ Isub :: nil
  | AMult a1 a2 => compile_aexp a1 ++ compile_aexp a2 ++ Imul :: nil
  end.
```

5.1.4. Evaluación booleana

Requisitos

La evaluación booleana de las expresiones en IMP necesitan responder a:

- Mantener el orden del código en IMP.
- Capacidad de evaluar un valor booleano básico.
- Capacidad de evaluar condiciones lógicas sobre valores.
- Capacidad de realizar el correspondiente salto de instrucciones en función de condiciones.

Diseño

Para responder a las necesidades anteriores, la evaluación del código en IMP queda reflejada como:

- Se construirá como un tipo recursivo para respetar el orden secuencial de ejecución del código.
- BTrue: Que se evalúa como un valor true.
- BFalse: Que se evalúa como un valor false.
- BEq: Que se evalúa como un salto condicional de instrucciones en función de la igualdad o desigualdad de las expresiones aritméticas.
- BLe: Que se evalúa como un salto condicional de instrucciones en función de la condición "menor o igual que" o "mayor que" de las expresiones aritméticas.
- BNot: Que se evalúa como la negación de una expresión booleana.
- BAnd: Que se evalúa como la operación lógica AND de dos expresiones booleanas.

Implementación

Por lo tanto la compilación de las expresiones booleanas queda definida como:

```
   Fixpoint compile_bexp (b: bexp) (cond: bool) (ofs: nat) : code :=
  match b with
  | BTrue =>
      if cond then Ibranch_forward ofs :: nil else nil
```

```
  | BFalse =>
      if cond then nil else Ibranch_forward ofs :: nil
  | BEq a1 a2 =>
      compile_aexp a1 ++ compile_aexp a2 ++
      (if cond then Ibeq ofs :: nil else Ibne ofs :: nil)
  | BLe a1 a2 =>
      compile_aexp a1 ++ compile_aexp a2 ++
      (if cond then Ible ofs :: nil else Ibgt ofs :: nil)
  | BNot b1 =>
      compile_bexp b1 (negb cond) ofs
  | BAnd b1 b2 =>
      let c2 := compile_bexp b2 cond ofs in
      let c1 := compile_bexp b1 false (if cond then length c2 else ofs + length c2) in
      c1 ++ c2
  end.
```

5.1.5. Evaluación de comandos

Requisitos

La evaluación de los comandos de IMP necesitan responder a:

- Mantener el orden del código en IMP.
- Capacidad de evaluar el salto de instrucción.
- Capacidad de evaluar la asignación de variables.
- Capacidad de evaluar la ejecución secuencial de código.
- Capacidad de evaluar un bucle If.
- Capacidad de evaluar un bucle While.

Diseño

Para responder a las necesidades anteriores, la evaluación del código en IMP queda reflejada como:

- Se construirá como un tipo recursivo para respetar el orden secuencial de ejecución del código.
- SKIP: Que no realiza ninguna ejecución sobre el código.
- (id ::= a): Que realiza una asignación de variable en función del id de la variable.
- (c1; c2): Que realiza una ejecución secuencial de la siguiente instrucción de código.
- IFB b THEN ifso ELSE ifnot FI: Que realiza un salto condicional del código en función de una expresión booleana tal y como lo haría una estructura de control if else.
- WHILE b DO body END: Que realiza un salto condicional del código en función de una expresión booleana tal y como lo haría una estructura de control while.

Implementación

Por lo tanto la compilación de los comandos queda definida como:

```
 Fixpoint compile_com (c: com) : code :=
match c with
| SKIP =>
    nil
| (id ::= a) =>
    compile_aexp a ++ Isetvar id :: nil
| (c1; c2) =>
    compile_com c1 ++ compile_com c2
| IFB b THEN ifso ELSE ifnot FI =>
    let code_ifso := compile_com ifso in
    let code_ifnot := compile_com ifnot in
    compile_bexp b false (length code_ifso + 1)
    ++ code_ifso
    ++ Ibranch_forward (length code_ifnot)
    :: code_ifnot
| WHILE b DO body END =>
    let code_body := compile_com body in
    let code_test := compile_bexp b false (length code_body + 1) in
    code_test
    ++ code_body
    ++ Ibranch_backward (length code_test + length code_body + 1)
    :: nil
end.
```

5.2– Módulo de optimización: Deadcode

Este módulo de optimización necesitará, en primer lugar, optimizar la eficacia del compilador. Existen muchas formas posibles de realizar esto en función de como se defina el concepto de optimización.

En este proyecto, se tratará de optimizar el funcionamiento de las variables. Ya que estas han sido construidas desde sus mismas bases y puede resultar interesante emular el comportamiento real de muchos optimizadores de variables en los compiladores.

Se utilizarán uno de los algoritmos de optimización de variables más utilizados en los compiladores actuales: un algoritmo de optimización de código muerto centrado en variables. Para ello se necesitará establecer un criterio de vivacidad para el código a compilar, así como herramientas para comparar variables y trabajar con conjuntos de ellos.

Esto permitirá al compilador reconocer que asignaciones de variables son totalmente prescindibles para el correcto funcionamiento del programa a compilar. Optimizando así el tiempo de compilación del programa y los recursos que maneja.

5.2.1. Comparador de variables

Requisitos

Los requisitos a satisfacer por el comparador de variables son:

- Capacidad para trabajar con conjuntos de variables.
- Capacidad para comparar variables.
- Capacidad para definir criterios de comparación entre variables.
- Capacidad para definir criterios de igualdad entre variables.
- Capacidad para instanciar conjuntos de variables.

Diseño

Para satisfacer estos requisitos, se ha decidido usar el conjunto de librerías estándar de Coq para trabajar con conjuntos finitos, en este caso, de variables. Para poder hacer uso de ellas, es necesario definir un módulo de ordenación y comparación para los identificadores de las variables.

En este caso reducido, se definen tan solo dos operaciones de comparación: la igualdad y el "menor que". Se ha decidido utilizar tan solo estos dos operadores ya que con solo estos se puede construir el conjunto de propiedades necesarias para comparar variables.

El conjunto de propiedades necesarias se han deducido a partir de lemas. Estas establecen propiedades necesarias de las operaciones de comparación como: reflexividad, simetría y transitividad.

Por último, se han especificado dos definiciones que permiten comparar dos identificadores de variables haciendo uso de las propiedades y lemas anteriores. Estas dos definiciones están demostradas lógicamente para asegurar su correcto comportamiento.

Implementación

Se define la implementación de este módulo junto con todas las propiedades necesarias:

```
  Module Id_Ordered <: OrderedType with Definition t := id.
Definition t := id.
Definition eq (x y: t) := x = y.
Definition lt (x y: t) := match x, y with Id nx, Id ny => nx < ny end.

Lemma eq_refl : forall x : t, eq x x.
Proof.
intro. reflexivity. Qed.

Lemma eq_sym : forall x y : t, eq x y -> eq y x.
Proof.
unfold eq; intros; auto. Qed.

Lemma eq_trans : forall x y z : t, eq x y -> eq y z -> eq x z.
Proof.
unfold eq; intros; congruence. Qed.
```

```
  Lemma lt_trans : forall x y z : t, lt x y -> lt y z -> lt x z.
Proof.
unfold lt; intros. destruct x; destruct y; destruct z. omega. Qed.

  Lemma lt_not_eq : forall x y : t, lt x y -> ~ eq x y.
Proof.
    unfold lt, eq; intros. destruct x; destruct y.
    assert (n <> n0) by omega. congruence.
  Qed.
```

Se implementa las dos definiciones de comparación entre variables.

```
 Definition compare: forall (x y: t), Compare lt eq x y.
Proof.
    intros. case x; intro nx. case y; intro ny.
    remember (beq_nat nx ny). destruct b.
    apply EQ. red. f_equal. apply beq_nat_true. auto.
    assert (nx <> ny). apply beq_nat_false. auto.
    remember (ble_nat nx ny). destruct b.
    assert (nx <= ny). apply ble_nat_true; auto.
    apply LT. red. omega.
    assert (~(nx <= ny)). apply ble_nat_false; auto.
    apply GT. red. omega.
  Defined.

  Definition eq_dec: forall (x y: t), {x=y} + {x<>y}.
Proof.
    intros; destruct x; destruct y.
    remember (beq_nat n n0); destruct b.
    left. f_equal. apply beq_nat_true. auto.
    right. assert (n <> n0). apply beq_nat_false; auto. congruence.
  Defined.
End Id_Ordered.
```

Por último, se instancian los módulos de Coq para operar con conjuntos en función del módulo de comparación anteriormente creado:

```
    Module VS := FSetAVL.Make(Id_Ordered).
    Module VSP := FSetProperties.Properties(VS).
    Module VSdecide := FSetDecide.Decide(VS).
    Import VSdecide.
```

5.2.2. Evaluación de las variables

Requisitos

La evaluación de variables responde a los siguientes requisitos:

- Capacidad de mantener la secuencialidad de las variables.
- Capacidad para evaluar las variables de una expresión aritmética.
- Capacidad para evaluar las variables de una expresión booleana.
- Capacidad para evaluar las variables en comandos de programa.
- Capacidad para agrupar las variables en conjuntos.

Diseño

Para satisfacer los requisitos previos, se ha diseñado un conjunto de evaluadores para expresiones del compilador cuyas especificaciones se narran a continuación:

- Serán métodos recursivos para respetar la secuencialidad y el orden de aparición de las variables en el código original.
- Se realizará un método evaluador para expresiones aritméticas.
- Se realizará un método evaluador para expresiones booleanas.
- Se realizará un método evaluador para expresiones que reflejen comandos del programa.
- Las variables se agruparán en conjuntos: VS.empty (conjunto vacío) VS.Union (Unión de conjuntos) y VS.singleton (conjunto unitario).

Implementación

La implementación para estos métodos evaluadores en función de la expresión es la siguiente:

```
Fixpoint fv_aexp (a: aexp) : VS.t :=
match a with
| ANum n => VS.empty
| AId v => VS.singleton v
| APlus a1 a2 => VS.union (fv_aexp a1) (fv_aexp a2)
| AMinus a1 a2 => VS.union (fv_aexp a1) (fv_aexp a2)
| AMult a1 a2 => VS.union (fv_aexp a1) (fv_aexp a2)
end.

Fixpoint fv_bexp (b: bexp) : VS.t :=
match b with
| BTrue => VS.empty
| BFalse => VS.empty
| BEq a1 a2 => VS.union (fv_aexp a1) (fv_aexp a2)
| BLe a1 a2 => VS.union (fv_aexp a1) (fv_aexp a2)
| BNot b1 => fv_bexp b1
| BAnd b1 b2 => VS.union (fv_bexp b1) (fv_bexp b2)
end.

Fixpoint fv_com (c: com) : VS.t :=
match c with
| SKIP => VS.empty
| x ::= a => fv_aexp a
| (c1; c2) => VS.union (fv_com c1) (fv_com c2)
| IFB b THEN c1 ELSE c2 FI => VS.union (fv_bexp b) (VS.union (fv_com c1) (fv_com c2))
| WHILE b DO c END => VS.union (fv_bexp b) (fv_com c)
end.
```

5.2.3. Analizador de vivacidad de las variables

Requisitos

El analizador de vivacidad de las variables se encargará de clasificar los conjuntos de variables en vivas o muertas. responderá a dichas necesidades:

- Capacidad para clasificar las variables en conjuntos.
- Capacidad para reconocer asignaciones de variables.
- Capacidad para reconocer estructuras de control y evaluar la vivacidad de sus variables.
- Respetar el tratamiento secuencial del código.

Diseño

En cuanto al diseño, se han escogido esta serie de elecciones:

- Se creará un método live que evaluará la vivacidad de las asignaciones de variables.
- El método live será un método recursivo para respetar la secuencialidad del código.
- Se evaluará cada posible comando del código y se clasificarán sus asignaciones de variables en conjuntos.
- En la asignación de variables, se estudiará si dicha asignación se considera viva o muerta.

Implementación

La implementación del algoritmo live viene definida a continuación junto con algunas demostraciones interesantes para su uso:

```
  Fixpoint live (c: com) (L: VS.t) : VS.t :=
match c with
    | SKIP => L
    | x ::= a =>
        if VS.mem x L
        then VS.union (VS.remove x L) (fv_aexp a)
        else L
    | (c1; c2) =>
        live c1 (live c2 L)
    | IFB b THEN c1 ELSE c2 FI =>
        VS.union (fv_bexp b) (VS.union (live c1 L) (live c2 L))
    | WHILE b DO c END =>
        let L' := VS.union (fv_bexp b) L in
        let default := VS.union (fv_com (CWhile b c)) L in
        fixpoint (fun x => VS.union L' (live c x)) default
end.

Lemma live_upper_bound:
  forall c L,
  VS.Subset (live c L) (VS.union (fv_com c) L).
    Proof.
```

```
    induction c; intros; simpl.
    fsetdec.
    case_eq (VS.mem i L); intros. fsetdec. fsetdec.
    generalize (IHc1 (live c2 L)). generalize (IHc2 L). fsetdec.
    generalize (IHc1 L). generalize (IHc2 L). fsetdec.
    apply fixpoint_upper_bound. intro x. generalize (IHc x). fsetdec.
  Qed.

Lemma live_while_charact:
  forall b c L,
  let L' := live (WHILE b DO c END) L in
  VS.Subset (fv_bexp b) L' /\ VS.Subset L L' /\ VS.Subset (live c L') L'.
  Proof.
    intros.
    generalize (fixpoint_charact
      (fun x : VS.t => VS.union (VS.union (fv_bexp b) L) (live c x))
            (VS.union (VS.union (fv_bexp b) (fv_com c)) L)).
    simpl in L'. fold L'. intros [A|A].
    split. generalize A; fsetdec. split; generalize A; fsetdec.
    split. rewrite A. fsetdec.
    split. rewrite A. fsetdec.
    eapply VSP.subset_trans. apply live_upper_bound. rewrite A. fsetdec.
  Qed.
```

5.2.4. Analizador de código muerto

Requisitos

Los requisitos necesarios para el analizador de código muerto son:

- Capacidad de analizar los comandos del programa y reconocer asignaciones de variables
- Capacidad de sustituir las asignaciones de variables por algún estado que no afecte a la correcta ejecución del programa
- Capacidad de reconocer las variables muertas y vivas.
- Capacidad de sustituir solo las variables muertas.
- Capacidad de respetar la naturaleza secuencial del código.

Diseño

Para ello, se ha generado un algoritmo llamado dce que se encargará de responder a dichos requisitos con estas decisiones de diseño

- Se ha definido como un método recursivo para respetar la naturaleza secuencial del código.
- Se han definido los distintos estados para cada uno de los comandos del código.
- El estado de asignación de variables se encarga de comprobar si la variable está en el conjunto de las vivas. Si no, se reemplaza la asignación por un comando SKIP.

Implementación

La implementación del algoritmo dce es la siguiente:

```
 Fixpoint dce (c: com) (L: VS.t): com :=
match c with
| SKIP => SKIP
| x ::= a => if VS.mem x L then x ::= a else SKIP
| (c1; c2) => (dce c1 (live c2 L); dce c2 L)
| IFB b THEN c1 ELSE c2 FI => IFB b THEN dce c1 L ELSE dce c2 L FI
| WHILE b DO c END => WHILE b DO dce c (live (WHILE b DO c END) L) END
end.
```

5.3– Módulo de optimización: Regalloc

Este módulo trabajará de forma complementaria al módulo de optimización anterior, ya que fija su objetivo en el mismo elemento del código: las variables. En este caso, el módulo de optimización Regalloc.

Los requisitos a cumplir por este nuevo sistema son bastante parecido a los del módulo anterior. El objetivo será aumentar el rendimiento del programa al relocalizar los datos más utilizados por el programa. Esto permitirá un aumento de rapidez a la hora de ejecutar el programa. Y una optimización más eficaz de los recursos en memoria.

Para cumplir con estos requisitos se propone utilizar un algoritmo que realizará un estudio sobre las variables para decidir de entre ellas el mayor número de variables que pueden desplazarse a una posición de memoria privilegia en los registros del procesador. Por cuestión de simplicidad al emular este comportamiento real, en nuestro caso la relocalización se representará como un renombre en las variables que estarían en registros del procesador.

5.3.1. Evaluación de expresiones

Requisitos

Los requisitos a cumplir en la evaluación de expresiones son:

- Capacidad para reconocer cuando una expresión aritmética es una asignación de variables.
- Capacidad para renombrar una variable.
- Capacidad para evaluar las expresiones aritméticas.
- Capacidad para evaluar las expresiones booleanas.

Diseño

El diseño se explicita como:

- Se ha definido un método: expr_is_var. Que se encarga de evaluar cuando una expresión es una asignación de variables.
- Se devolverá un option en expr_is_var para garantizar que el método devuelve tipos correctos cuando no encuentra una asignación de variables.

- Se ha definido un método: f. Que se encarga de renombrar una variable.
- Los métodos de evaluación serán métodos inductivos ya que no se necesitan tipos recursivos. Facilitando su tratamiento.
- Se ha definido un método de evaluación para expresiones aritméticas. Cuando se produce una asignación, se realiza el renombrado de la variable.
- Se ha definido un método de evaluación para expresiones booleanas. En el caso de que en una expresión booleana se encuentren expresiones aritméticas.

Implementación

La implementación de los métodos expr_is_var y f es la siguiente:

```
  Definition expr_is_var (a: aexp): option id :=
match a with AId x => Some x | _ => None end.

Lemma expr_is_var_correct:
  forall a x, expr_is_var a = Some x -> a = AId x.
Proof.
  unfold expr_is_var; intros. destruct a; congruence.
Qed.

  Variable f: id -> id.
```

Para los métodos de evaluación, la implementación queda como:

```
  Fixpoint rename_aexp (a: aexp) : aexp :=
match a with
| ANum n => ANum n
| AId x => AId (f x)
| APlus a1 a2 => APlus (rename_aexp a1) (rename_aexp a2)
| AMinus a1 a2 => AMinus (rename_aexp a1) (rename_aexp a2)
| AMult a1 a2 => AMult (rename_aexp a1) (rename_aexp a2)
end.

Fixpoint rename_bexp (b: bexp) : bexp :=
  match b with
  | BTrue => BTrue
  | BFalse => BFalse
  | BEq a1 a2 => BEq (rename_aexp a1) (rename_aexp a2)
  | BLe a1 a2 => BLe (rename_aexp a1) (rename_aexp a2)
  | BNot b1 => BNot (rename_bexp b1)
  | BAnd b1 b2 => BAnd (rename_bexp b1) (rename_bexp b2)
  end.
```

5.3.2. Transformación de los comandos

Requisitos

Los requisitos para la transformación de comandos son los siguientes:

- Capacidad para evaluar todos los posibles estados de los comandos del código.
- Capacidad para reconocer una asignación de variables.

- Capacidad para estudiar cuando una variable es candidata a ser renombrada.
- Capacidad para computar el resto del código en comandos de estructuras de control.

Diseño

Por lo tanto, el diseño queda como:

- La función será un método inductivo ya que no se necesitan tipos recursivos. Facilitando su tratamiento.
- La función se definirá para todos los estados de los comandos del código.
- Para la asignación de variables, se utilizarán los métodos de expr_is_var y f para reconocer cuando una variable tiene que ser renombrada.
- Para las estructuras de control, se utilizarán los métodos evaluativos para seguir evaluando el resto del código computable.

Implementación

```
Fixpoint transf_com (c: com) (L: VS.t): com :=
  match c with
  | SKIP => SKIP
  | x ::= a =>
      if VS.mem x L then
        match expr_is_var a with
        | None => (f x) ::= (rename_aexp a)
        | Some y => if beq_id (f x) (f y) then SKIP else (f x) ::= (rename_aexp a)
        end
      else SKIP
  | (c1; c2) =>
      (transf_com c1 (live c2 L); transf_com c2 L)
  | IFB b THEN c1 ELSE c2 FI =>
      IFB rename_bexp b THEN transf_com c1 L ELSE transf_com c2 L FI
  | WHILE b DO c END =>
      WHILE rename_bexp b DO transf_com c (live (WHILE b DO c END) L) END
end.
```

CAPÍTULO 6

Análisis temporal y costes

6.1– Análisis temporal del proyecto

Como fase previa al desarrollo del proyecto, incluso antes de que se decidiera la construcción de un compilador. Se estableció una estimación media para el tiempo requerido por cada fase a realizar.

Las fases en las que se divide la planificación del proyecto han sido: investigación, planificación, estudio teórico, desarrollo práctico, realización de la memoria y revisión y mejoras.

Para la fase de investigación, se dedicó el tiempo a investigar posibles temas de interés sobre los que realizar el proyecto. Buscando un equilibrio entre teoría y práctica que fuera adecuado para las horas exigidas y el tiempo disponible. En la fase de planificación se estructuró el desarrollo del proyecto a tratar, que partes destacar, que partes dejar fuera, que partes revisar y en general planificar a grandes rasgos el desarrollo no solo del caso práctico, si no de la totalidad del proyecto.

La fase de estudio teórico engloba todo el tiempo dedicado a reunir las bases teóricas y fundamentos necesarios para la realización del proyecto. Esta parte ha conformado el grueso del tiempo total debido a su densidad. La fase de desarrollo práctico recoge todo el tiempo dedicado al desarollo práctico de la teoría estudiada, en este caso, la creación de un compilador.

A continuación se incluye una tabla con la estimación en horas para cada fase y la cantidad real de horas depositadas:

Fase	Horas estimadas	Horas empleadas
Investigación	10	10
Planificación	2	1
Estudio teórico	150	190
Desarrollo práctico	200	150
Memoria	50	80
Revisión y mejoras	5	2

6.2– Estimación real: análisis temporal y costes

Para poner un poco de realidad en esta estimación se ha estudiado el coste aproximado tanto en tiempo como en dinero de la creación de un compilador. La cadena de herramientas de compilación es una de las más largas y complejas de cualquier sistema. Gracias al código abierto se ha podido realizar estimaciones del coste total de compiladores tan masivos como GCC o LLVM.

Basándose en parámetros algo más cuantificables, se estima que GCC tiene más o menos 5 millones de líneas de código. Mientras que LLVM son 1.6 millones, pero claro, un compilador real está compuesto por más herramientas que el compilador en sí. Por lo que a continuación se recoge una estimación en líneas de código de distintos componentes de compiladores:

- Debugger: GDB: 800 mil líneas. LLDB: 600 mil líneas.
- Linker: GNU Id: 160 mil líneas. gold: 140 mil líneas. lld: 60 mil líneas.
- Assembler/disassembler: GNU gas 850 mil líneas
- Binarios de utilidad: GNU: 90 mil líneas.
- Librerías de emulación: libgcc: 130 mil líneas. CompilerRT 340 mil líneas.

En la gran mayoría de herramientas los test de regresión están incluidos en las líneas de código principal. Excepto para LLVM, los cuales constituyen casi 500 mil líneas más.

6.2.1. Caso general

Se considera el caso general de implementación: la creación de una nueva arquitectura para un sistema Linux que necesita dar soporte a C y a C++. Esta arquitectura deberá proveer un gran rango de implementaciones, desde procesos básicos embebidos en el sistema hasta procesos más grandes de aplicaciones Linux.

La primera *release* de este sistema de herramientas de compilación puede llevar de 1 a 3 años de esfuerzos en I+D. La primera prueba conceptual debería estar terminada para los siguientes 3 meses. La implementación de todas las funcionalidades necesarias puede llevar de 6 a 9 meses.

Los tests de producción como mínimo serán 6 meses más. Pero con un sistema tan complejo como este pueden llevar hasta 12 meses. La primera prueba de usuarios sumaría 3 meses más hasta poder alcanzar un número más grande de usuarios a testear que podría ser 9 meses más.

Para un proyecto así se necesitaría un equipo completo de ingenieros: Especialistas en compiladores, expertos debuggers, ingenieros de librerías y demás campos concretos necesarios. Además, hay que tener en cuenta que la ingeniería de compiladores es una de las ingenierías en el campo de la computación más demandantes a nivel técnico. E incluso así, puede no ser experto en todas las cualidades necesarias para la creación de un compilador.

6.2.2. Caso simple

Habiendo considerado el caso más general, en el que se necesita construir todo desde cero. También puede suceder que no se necesite construir todas las partes de las herramientas de compilación mencionadas anteriormente. Ya que hay numerosas aplicaciones que gestionan dichas partes e incluso subcontratan su implementación y desarrollo. Existen también numerosas propuestas comerciales para cubrir dichas necesidades

Para casos así, la prueba conceptual puede llevar 3 meses también. Sin embargo, la implementación y desarrollo de las funcionalidades puede recortarse hasta un tiempo de 3 a 5 meses. Los tests de producción siguen siendo un grueso del tiempo, con 6 meses de estimación. Sin embargo se puede acortar la base de usuarios de pruebas con 3 meses tan solo.

6.2.3. Caso de estudio

En 2016, a la empresa Embecosm se le encomendó la tarea de crear una base de herramientas basada en LLVM capaz de compilar sus codecs en C y conseguir código de alta calidad. Con la premisa de que este código de alta calidad pudiera ser modificado dado el caso. La empresa del cliente ya disponía de ciertos elementos del compilador, como el linker y el assembler. Para el cliente era importante el conocer el proceso de creación del compilador, por lo que uno de sus ingenieros se unió al equipo de implementación y estuvo dando soporte al compilador hasta el final del proyecto.

En los primeros 3 meses se construyó el conjunto de herramientas basadas en el assembler y linker ya existentes. Para la fase dos, se implementó un dissasembler basado en GNU usando CGEN, lo que llevó aproximadamente 10 días. Con estas dos características ya implementadas el trabajo se realizó de manera mucho más rápida, lo que implicó dedicar más tiempo a los tests de regresión para la suite de herramientas.

Debido a este caso concreto, en el que ya se disponían de ciertos elementos necesarios para construir el compilador. Muchos de los esfuerzos finales se destinaron a procesos de optimización como recolectores de basura y optimizadores de memoria. Lo cual supuso una inversión ingenieril de 120 días. El cliente decidió que tenían todas las funcionalidades necesarias y que no era necesario realizar nada más.

Dos factores clave supusieron la creación de una suite de herramientas de compilación en tan poco tiempo:

- Usar código abierto para el compilador: Las herramientas usadas en el proyecto son fruto de un esfuerzo acumulado por miles de ingenieros durante años y décadas dedicadas a la tecnología de compilación.
- Un equipo de expertos en compilación: A pesar del corto tiempo de desarrollo, un equipo de 5 ingenieros fue necesario para realizarlo. Cada uno con años de experiencia en el campo. Ningún individuo podría conocer sobre emulación, GDB, assemblers CGEN, linkers GNU y sobre el compilador LLVM.

Para concretizar el análisis a nivel de costes de un compilador se relata un ejemplo de estimación para el compilador LLVM.

Basándose en la última versión de LLVM (8.0.1) hasta la fecha revisada. LLVM es el resultado de 81.033.801 líneas de código añadidas y 22.048.147 líneas borradas. Utilizando el método SLOCCount de David A. Wheeler para conocer el número de líneas de código útiles y un coste estimado de su desarrollo basado en el modelo COCOMO, obtenemos que LLVM 8.0.1 está formado por 6.887.487 líneas de código. Con un coste de 529.895.190$

Según la estimación realizada por Chris Cummins en su tesis, esta sería la estimación de los 10 compiladores más populares:

Project	Started	Developers	LOC	Estimated Cost
FreePascal	2005	54	3,845,685	$198,619,187
GCC 9.2.0	1988	617	5,591,759	$425,747,278
Glasgow Haskell Compiler 8.8.1	2001	1,119	761,097	$52,449,098
Intel Graphics Compiler 1.0.10	2018	149	684,688	$46,934,626
LLVM 8.0.1	2001	1,210	6,887,489	$529,895,190
OpenJDK 14+10	2007	883	7,955,827	$616,517,786
Roslyn .NET 16.2	2014	496	2,705,092	$198,619,187
Rust 1.37.0	2010	2,737	852,877	$59,109,425
Swift	2010	857	665,238	$45,535,689
v8 7.8.112	2008	736	3,048,793	$225,195,832

Figura 6.1: Estimación de costes. De izquierda a derecha: Proyecto, fecha de inicio, nº de desarrolladores, líneas de código, coste estimado.

CAPÍTULO 7

Comparación con otras alternativas

7.1– Intérprete

La primera comparación significativa a tener en cuenta puede ser la del uso de otra arquitectura a la hora de ejecutar código. Por lo tanto, se podría haber propuesto un intérprete en lugar de un compilador para el lenguaje IMP. Ambos realizan la misma función: Traducir código de alto nivel, entendible por los humanos, a un código de más bajo nivel entendible por la máquina. En nuestro caso concreto, a una máquina virtual.

En líneas generales, las principales diferencias entre intérpretes y compiladores son:

- Los compiladores pueden transformar toda la semántica del código fuente de una vez. Mientras que los intérpretes necesitan hacer conversiones intermedias cada vez que una instrucción o función es ejecutada.
- Ambos transforman el código fuente en tokens y ambos suelen generar árboles de análisis para sus estructuras. La diferencia es que en los sistemas compilados se genera código máquina independiente, mientras que en los sistemas interpretados realizan las acciones descritas por el programa de alto nivel.
- Por regla general, los intérpretes tardan menos tiempo en analizar el código fuente pero tardan mucho más tiempo en ejecutarlo.
- Los intérpretes interrumpen su traducción al encontrar el primer error. Mientras que los compiladores muestran todos los posibles errores del programa.
- Los compiladores generan un programa final. Que puede ser ejecutado independientemente del programa original. Los intérpretes no generan ningún programa final independiente. Por lo que necesitan traducir el programa original cada vez que se ejecuta.

7.2– Código máquina

Otro posible cambio en la arquitectura podría haber sido la de traducir directamente el código de IMP a lenguaje máquina, prescindiendo completamente de la máquina virtual. Para empezar, es necesario entender el por qué de las máquinas virtuales. Estas surgen como respuesta a la problemática portabilidad de código. Ya que había que crear compiladores de lenguajes de alto nivel para cada arquitectura hardware. Por lo tanto la compilación dependía fuertemente del hardware adyacente.

Al dividir la compilación en dos procesos: De traducir el lenguaje de alto nivel a un lenguaje intermedio. Y de este al lenguaje máquina, se consiguió sobreponer dicha dependencia. La idea es que esta compilación intermedia abstraiga al lenguaje de alto nivel del hardware, ya que la máquina virtual actúa como valor de entrada para el compilador a lenguaje máquina. Por lo tanto tan solo hay que traducir los lenguajes de alto nivel a este lenguaje intermedio. En nuestro caso, el subconjunto de instrucciones de la JVM.

7.3– Máquina virtual

Sabiendo lo anterior, se podría haber usado otro lenguaje intermedio (o máquina virtual) para compilar el lenguaje de alto nivel. En el mercado otra alternativa muy interesante es la *Common Languaje Runtine* (CLR). Esta es la máquina virtual del framework de Microsoft .NET Framework. La gran diferencia entre estas dos máquina virtuales serían el conjunto de instrucciones utilizado para evaluar las expresiones. Ya que la JVM está construida para convertir el bytecode en lenguaje máquina. Mientras que la CLR no realiza una traducción directa, si no que provee del entorno necesario para ejecutar los programas traducidos.

7.3.1. Implementación de la JVM

Dentro incluso de que se decida utilizar la JVM existen diferentes implementaciones de la misma. En este proyecto se ha seguido la implementación principal de la JVM HotSpot producida por Oracle. Pero a continuación se recogen otras implementaciones actualmente activas y de código abierto:

- Codename One
- Eclipse OpneJ9
- GraalVM
- Jikes RVM
- leJOS
- Maxine

CAPÍTULO 8

Pruebas

8.1– Preservación semántica: Compil, Big-step coinductivos

8.1.1. Lemas necesarios

Para empezar a comprobar la equivalencia semántica de los programas IMP y las compilaciones resultantes primero se necesita considerar secuencias de código Cx. Estas se encuentran en la posición pc y están contenidas en una secuencia mayor de código de tal forma que C = c1 ++ c2 ++ c3. El siguiente predicado codeesq_at C pc C2 introduce esto:

```
Inductive codeseq_at: code -> nat -> code -> Prop :=
  | codeseq_at_intro: forall C1 C2 C3 pc,
      pc = length C1 ->
      codeseq_at (C1 ++ C2 ++ C3) pc C2.
```

También se necesitan una serie de lemas muy básicos acerca del método code_at y codeseq_at:

```
Lemma code_at_app:
  forall i c2 c1 pc,
  pc = length c1 ->
  code_at (c1 ++ i :: c2) pc = Some i.
Proof.
  induction c1; simpl; intros; subst pc; auto.
Qed.

Lemma codeseq_at_head:
  forall C pc i C',
  codeseq_at C pc (i :: C') ->
  code_at C pc = Some i.
Proof.
  intros. inv H. simpl. apply code_at_app. auto.
Qed.

Lemma codeseq_at_tail:
  forall C pc i C',
  codeseq_at C pc (i :: C') ->
```

```
  codeseq_at C (pc + 1) C'.
Proof.
  intros. inv H.
  change (C1 ++ (i :: C') ++ C3)
    with (C1 ++ (i :: nil) ++ C' ++ C3).
  rewrite <- app_ass. constructor. rewrite app_length. auto.
Qed.

Lemma codeseq_at_app_left:
  forall C pc C1 C2,
  codeseq_at C pc (C1 ++ C2) ->
  codeseq_at C pc C1.
Proof.
  intros. inv H. rewrite app_ass. constructor. auto.
Qed.

Lemma codeseq_at_app_right:
  forall C pc C1 C2,
  codeseq_at C pc (C1 ++ C2) ->
  codeseq_at C (pc + length C1) C2.
Proof.
  intros. inv H. rewrite app_ass. rewrite <- app_ass. constructor.
  rewrite app_length. auto.
Qed.

Lemma codeseq_at_app_right2:
  forall C pc C1 C2 C3,
  codeseq_at C pc (C1 ++ C2 ++ C3) ->
  codeseq_at C (pc + length C1) C2.
Proof.
  intros. inv H. repeat rewrite app_ass. rewrite <- app_ass. constructor.
  rewrite app_length. auto.
Qed.
```

8.1.2. Verificación para expresiones aritméticas

Se recuerda los requisitos para el código generado para una expresión artimética a. Esta debe:

- Se ejecuta de manera secuencial.
- Introduce el valor de a al inicio de la pila.
- Mantiene el estado de la variable.

Por lo tanto, se comprueba que el código que genera el método de compilación compile_aexp cumple con lo exigido:

```
Lemma compile_aexp_correct:
  forall C st a pc stk,
  codeseq_at C pc (compile_aexp a) ->
  star (transition C)
       (pc, stk, st)
       (pc + length (compile_aexp a), aeval st a :: stk, st).
```

```
Proof.
  induction a; simpl; intros.

Case "ANum".
  apply star_one. apply trans_const. eauto with codeseq.

Case "AId".
  apply star_one. apply trans_var. eauto with codeseq.

Case "APlus".
  eapply star_trans.
  apply IHa1. eauto with codeseq.
  eapply star_trans.
  apply IHa2. eauto with codeseq.
  apply star_one. normalize. apply trans_add. eauto with codeseq.

Case "AMinus".
  eapply star_trans.
  apply IHa1. eauto with codeseq.
  eapply star_trans.
  apply IHa2. eauto with codeseq.
  apply star_one. normalize. apply trans_sub. eauto with codeseq.

Case "AMult".
  eapply star_trans.
  apply IHa1. eauto with codeseq.
  eapply star_trans.
  apply IHa2. eauto with codeseq.
  apply star_one. normalize. apply trans_mul. eauto with codeseq.
Qed.
```

Queda así verificada la equivalencia semántica de la compilación de expresiones aritméticas con su respectiva explicitación en IMP.

8.1.3. Verificación para expresiones booleanas

Para las expresiones booleanas, también se recuerdan los requisitos exigidos:

- Salta adelante *ofs* instrucciones cuando se evalúa como una condición booleana, *cond*, true.
- Ejecuta secuencialmente si b se evalúa como la negación de *cond*
- Deja la pila y el estado sin cambios.

Se aporta la verifiación de que se mantiene la correctitud en la semántica de las expresiones booleanas compiladas:

```
emma compile_bexp_correct:
  forall C st b cond ofs pc stk,
  codeseq_at C pc (compile_bexp b cond ofs) ->
  star (transition C)
       (pc, stk, st)
       (pc + length (compile_bexp b cond ofs) +
```

```
       if eqb (beval st b) cond then ofs else 0, stk, st).
Proof.
  induction b; simpl; intros.

Case "BTrue".
  destruct cond; simpl.
  SCase "BTrue, true".
    apply star_one. apply trans_branch_forward with ofs. eauto with codeseq. auto.
  SCase "BTrue, false".
    repeat rewrite plus_0_r. apply star_refl.

Case "BFalse".
  destruct cond; simpl.
  SCase "BFalse, true".
    repeat rewrite plus_0_r. apply star_refl.
  SCase "BFalse, false".
    apply star_one. apply trans_branch_forward with ofs. eauto with codeseq. auto.

Case "BEq".
  eapply star_trans.
  apply compile_aexp_correct with (a := a). eauto with codeseq.
  eapply star_trans.
  apply compile_aexp_correct with (a := a0). eauto with codeseq.
  apply star_one. normalize.
  destruct cond.
  SCase "BEq, true".
    apply trans_beq with ofs. eauto with codeseq.
    destruct (beq_nat (aeval st a) (aeval st a0)); simpl; omega.
  SCase "BEq, false".
    apply trans_bne with ofs. eauto with codeseq.
    destruct (beq_nat (aeval st a) (aeval st a0)); simpl; omega.

Case "BLe".
  eapply star_trans.
  apply compile_aexp_correct with (a := a). eauto with codeseq.
  eapply star_trans.
  apply compile_aexp_correct with (a := a0). eauto with codeseq.
  apply star_one. normalize.
  destruct cond.
  SCase "BLe, true".
    apply trans_ble with ofs. eauto with codeseq.
    destruct (ble_nat (aeval st a) (aeval st a0)); simpl; omega.
  SCase "BLe, false".
    apply trans_bgt with ofs. eauto with codeseq.
    destruct (ble_nat (aeval st a) (aeval st a0)); simpl; omega.

Case "BNot".
  replace (eqb (negb (beval st b)) cond)
     with (eqb (beval st b) (negb cond)).
  apply IHb; auto.
  destruct (beval st b); destruct cond; auto.
```

```
Case "BAnd".
  set (code_b2 := compile_bexp b2 cond ofs) in *.
  set (ofs' := if cond then length code_b2 else ofs + length code_b2) in *.
  set (code_b1 := compile_bexp b1 false ofs') in *.
  apply star_trans with (pc + length code_b1 + (if eqb (beval st b1)
                                    false then ofs' else 0), stk, st).
  apply IHb1. eauto with codeseq.
  destruct cond.
  SCase "BAnd, true".
    destruct (beval st b1); simpl.
    SSCase "b1 evaluates to true".
      normalize. rewrite plus_0_r. apply IHb2. eauto with codeseq.
    SSCase "b1 evaluates to false".
      normalize. rewrite plus_0_r. apply star_refl.
  SCase "BAnd, false".
    destruct (beval st b1); simpl.
    SSCase "b1 evaluates to true".
      normalize. rewrite plus_0_r. apply IHb2. eauto with codeseq.
    SSCase "b1 evaluates to false".
      replace ofs' with (length code_b2 + ofs). normalize. apply star_refl.
      unfold ofs'; omega.
Qed.
```

8.1.4. Verificación para comandos: caso terminativo

Para los comandos que realizan una compilación terminativa, es decir, que alcanzan la compilación final como una sucesión de reducciones finita, se explicita su verificación semántica:

```
Lemma compile_com_correct_terminating:
  forall C st c st',
  c / st ==> st' ->
  forall stk pc,
  codeseq_at C pc (compile_com c) ->
  star (transition C)
       (pc, stk, st)
       (pc + length (compile_com c), stk, st').
Proof.
  induction 1; intros stk pc AT.

Case "SKIP".
  simpl in *. rewrite plus_0_r. apply star_refl.

Case ":=".
  simpl in *. subst n.
  eapply star_trans. apply compile_aexp_correct. eauto with codeseq.
  apply star_one. normalize. apply trans_setvar. eauto with codeseq.

Case "sequence".
  simpl in *.
  eapply star_trans. apply IHceval1. eauto with codeseq.
```

```
  normalize. apply IHceval2. eauto with codeseq.

Case "if true".
  simpl in *.
  set (code1 := compile_com c1) in *.
  set (codeb := compile_bexp b1 false (length code1 + 1)) in *.
  set (code2 := compile_com c2) in *.
  eapply star_trans.
  apply compile_bexp_correct with (b := b1) (cond := false) (ofs := length code1 + 1).
  eauto with codeseq.
  rewrite H. simpl. rewrite plus_0_r. fold codeb. normalize.
  eapply star_trans. apply IHceval. eauto with codeseq.
  apply star_one. eapply trans_branch_forward. eauto with codeseq. omega.

Case "if false".
  simpl in *.
  set (code1 := compile_com c1) in *.
  set (codeb := compile_bexp b1 false (length code1 + 1)) in *.
  set (code2 := compile_com c2) in *.
  eapply star_trans.
  apply compile_bexp_correct with (b := b1) (cond := false) (ofs := length code1 + 1).
  eauto with codeseq.
  rewrite H. simpl. fold codeb. normalize.
  replace (pc + length codeb + length code1 + S(length code2))
     with (pc + length codeb + length code1 + 1 + length code2).
  apply IHceval. eauto with codeseq. omega.

Case "while false".
  simpl in *.
  eapply star_trans.
  apply compile_bexp_correct with (b := b1) (cond := false)
                                             (ofs := length (compile_com c1) + 1).
  eauto with codeseq.
  rewrite H. simpl. normalize. apply star_refl.

Case "while true".
  apply star_trans with (pc, stk, st').
  simpl in *.
  eapply star_trans.
  apply compile_bexp_correct with (b := b1) (cond := false)
                                            (ofs := length (compile_com c1) + 1).
  eauto with codeseq.
  rewrite H; simpl. rewrite plus_0_r.
  eapply star_trans. apply IHceval1. eauto with codeseq.
  apply star_one.
  eapply trans_branch_backward. eauto with codeseq. omega.
  apply IHceval2. auto.
Qed.

Theorem compile_program_correct_terminating:
  forall c st st',
```

```
  c / st ==> st' ->
  mach_terminates (compile_program c) st st'.
Proof.
```

8.1.5. Verificación para comandos: caso divergente

Se consideran una conjunto de estados máquina que corresponden a un caso divergente: PC apunta a compile_com y c diverge en el estado actual, como reflejan las semánticas big-step coinductivas:

```
Inductive diverging_state: code -> machine_state -> Prop :=
  | div_state_intro: forall c st C pc stk,
      c / st ==>  ->
      codeseq_at C pc (compile_com c) ->
      diverging_state C (pc, stk, st).
```

Se demuestra que desde cualquier estado divergente, la máquina siempre puede hacer una o varias transiciones para alcanzar otro estado divergente:

```
Lemma diverging_state_productive:
  forall C S1,
  diverging_state C S1 ->
  exists S2, plus (transition C) S1 S2 /\ diverging_state C S2.
Proof.
  intros. inv H. revert c st pc H0 H1.
  induction c; intros st pc EXECINF AT; inv EXECINF.
Case "seq left".
  simpl in AT. apply IHc1. auto. eauto with codeseq.
Case "seq right".
  simpl in AT.
  destruct (IHc2 st2 (pc + length (compile_com c1))) as [S [A B]]. auto.
  eauto with codeseq.
  exists S; split.
  eapply star_plus_trans. apply compile_com_correct_terminating.
  eauto. eauto with codeseq. apply A.
  apply B.
Case "if true".
  simpl in AT.
  set (code1 := compile_com c1) in *.
  set (codeb := compile_bexp b false (length code1 + 1)) in *.
  set (code2 := compile_com c2) in *.
  destruct (IHc1 st (pc + length codeb)) as [S [A B]]. auto. eauto with codeseq.
  exists S; split.
  eapply star_plus_trans.
  apply compile_bexp_correct with (b := b) (cond := false) (ofs := length code1 + 1).
  eauto with codeseq.
  rewrite H3; simpl. rewrite plus_0_r. apply A.
  apply B.
Case "if false".
  simpl in AT.
  set (code1 := compile_com c1) in *.
  set (codeb := compile_bexp b false (length code1 + 1)) in *.
  set (code2 := compile_com c2) in *.
```

```
  destruct (IHc2 st (pc + length codeb + length code1 + 1)) as [S [A B]]. auto.
  eauto with codeseq.
  exists S; split.
  eapply star_plus_trans.
  apply compile_bexp_correct with (b := b) (cond := false) (ofs := length code1 + 1).
  eauto with codeseq.
  rewrite H3; simpl. normalize. apply A.
  apply B.
Case "while body".
  simpl in AT.
  set (codec := compile_com c) in *.
  set (codeb := compile_bexp b false (length codec + 1)) in *.
  destruct (IHc st (pc + length codeb)) as [S [A B]]. auto. eauto with codeseq.
  exists S; split.
  eapply star_plus_trans.
  apply compile_bexp_correct with (b := b) (cond := false) (ofs := length codec + 1).
  eauto with codeseq.
  rewrite H1; simpl. rewrite plus_0_r. apply A.
  apply B.
Case "while loop".
  exists (pc, stk, st'); split.
  simpl in AT.
  set (codec := compile_com c) in *.
  set (codeb := compile_bexp b false (length codec + 1)) in *.
  eapply star_plus_trans.
  apply compile_bexp_correct with (b := b) (cond := false) (ofs := length codec + 1).
  eauto with codeseq.
  rewrite H1; simpl. rewrite plus_0_r.
  eapply star_plus_trans.
  apply compile_com_correct_terminating. eauto. eauto with codeseq.
  apply plus_one.
  eapply trans_branch_backward. eauto with codeseq. fold codec; fold codeb. omega.
  econstructor. eauto. auto.
Qed.
```

Partiendo de esto, se deduce que la máquina hace el mismo número de transiciones infinitas que si empezara en un estado divergente:

```
Lemma compile_com_correct_diverging:
  forall c st C pc stk,
  c / st ==>  -> codeseq_at C pc (compile_com c) ->
  infseq (transition C) (pc, stk, st).
Proof.
  intros.
  apply infseq_coinduction_principle_2 with (X := diverging_state C).
  apply diverging_state_productive.
  econstructor; eauto.
Qed.

Theorem compile_program_correct_diverging:
  forall c st,
  c / st ==>  ->
  mach_diverges (compile_program c) st.
```

```
Proof.
  intros. unfold compile_program. red.
  apply compile_com_correct_diverging with (c := c). auto.
  apply codeseq_at_intro with (C1 := nil); auto.
Qed.
```

CAPÍTULO 9

Conclusiones y desarrollos futuros

9.1– Conclusiones

Gracias a este proyecto se ha indagado en conceptos y tecnologías que hasta ahora se antojaban difusas y complejas. Se ha recorrido la construcción de un lenguaje de programación estableciendo sus bases sintácticas y semánticas. Para luego construir un compilador que mantuviera dicha semántica. Emulando el comportamiento de los compiladores que se usan a diario para crear cualquier software. Se ha llegado incluso a añadir estructuras de optimización a dicho compilador. Usando tecnologías actuales como analizadores estáticos de código, relocalización en memoria, optimización de los recursos.

Como competencias transversales se ha podido indagar en la programación funcional, en la especificación formal de programas y estructuras mediante su definición lógica. En el uso de herramientas como demostradores de teoremas tales como Coq. Se han explorado un gran conjunto de herramientas, conocimientos y tecnologías que se usan actualmente de manera agnóstica. Por lo cual parecen ni siquiera existir siendo las bases del mundo software.

Se ha podido indagar en el conocimiento de uno de los sistemas más complejos de la computación, utilizado a diario por particulares, empresas y productos. Uno de los logros tecnológicos más increíbles del campo de la computación. Se ha podido conocer su contexto histórico, su evolución, la justificación de sus partes. La labor de los compiladores es una titánica. Realizando la tarea de traducir nuestras ordenes y designios a la máquina para resolver problemas o satisfacer necesidades.

Se han quedado muchos conceptos teóricos más avanzados, tecnologías a usar y herramientas sin atención. Todo esto debido al corto tiempo disponible para investigar, indagar y reflejarlo en este proyecto. Un recurso que se considera esencial para esta temática es el conjunto de libros de *Software Foundations* por Benjamin C. Pierce, que se encuentra en la bibliografía. Constituye una fuente de conocimientos sobre el campo esencial, con explicaciones claras, ejemplos, puestas en práctica, etc. La colección de 6 libros hasta el momento es una joya del conocimiento libre.

9.2– Propuestas de mejora

Como se ha podido ver en el análisis de costes y temporal, justamente en la estimación real. Este proyecto engloba tecnologías muy concretas acerca de la creación de un compilador. Intentando siempre mantener un tono cercano en cuanto a lo que se ve en el ámbito profesional actual.

Debido al entorno en el que se enmarca este proyecto, a las horas estimadas, y al personal disponible, la envergadura y complejidad del mismo es proporcional a estas condiciones. En una situación totalmente real se podría haber indagado más no solo en la arquitectura de un compilador. Si no en el conjunto de herramientas de las que forma parte. Construyendo y definiendo assemblers y dissasembers, linkers, librerías útiles y demás herramientas necesarias en el complejo sistema de la compilación.

Para ello sería necesario disponer de un complejo equipo de profesionales expertos en la materia, un buen presupuesto, debido a que tareas como esta con gran complejidad y una gran carga ingenieril suelen ser costosas. También sería necesario un más que profundo conocimiento de las bases teóricas que conforman el grueso de herramientas necesarias. En la realidad, sería un proyecto que podría extenderse años y con un equipo muy numeroso de expertos cualificados.

Este trabajo intenta poner un pie en dicho ámbito y simplificar los procesos para permitir un análisis intuitivo y entendible del mismo. Se ha focalizado en el concepto de compilador para abarcar la parte más significativa de todo el proceso. Pero como se ha mencionando anteriormente, existen muchas herramientas que forman parte de este proceso. Con una gran complejidad constructiva detrás.

9.3– Futuros desarrollos:

Como posibles desarrollos futuros, se explicitan:

- Construcción de un intérprete.
- Conjunto de instrucciones distinto.
- Implementación diferente de la VM.
- Conjunto de operaciones sobre el lenguaje IMP más complejo.
- Optimización del compilador.
- Optimización de la memoria usada.
- Conjunto semántico distinto.
- Demostración de nuevas propiedades sobre el compilador.
- Uso de otras máquinas virtuales (CLR. .NET Framework).

Bibliografía

Appel, A. W. (2022). Coq standard library. Referencia: https://www.cs.princeton.edu/courses/archive/fall07/cos595/stdlib/html/Coq.FSets.FSetAVL.html (fecha de consulta: 2 de Enero de 2022)

Bennett, J. (2018). How much does a compiler cost? Referencia: https://www.embecosm.com/2018/02/26/how-much-does-a-compiler-cost/ (fecha de consulta: 12 de Mayo de 2022)

Casasola, R. A. G. (2022). Repositorio de código. Referencia: https://github.com/roG0d/Compil (fecha de realización: 9 de Enero de 2022)

Chong, S. (2022). Small-steps semantics. Referencia: https://groups.seas.harvard.edu/courses/cs152/2016sp/lectures/lec02-induction.pdf (fecha de consulta: 7 de Enero de 2022)

Crichton, W. (2022). Languaje programming theory. Referencia: https://stanford-cs242.github.io/f19/ (fecha de consulta: 19 de Febrero de 2022)

Cummins, C. (2022). Top 10 popular compilers. Referencia: https://github.com/ChrisCummins/phd/tree/master/docs/thesis (fecha de consulta: 7 de Mayo de 2022)

Gilles. (2022). Big-steps/small-steps semantics. Referencia: https://cs.stackexchange.com/questions/43294/difference-between-small-and-big-step-operational-semantics (fecha de consulta: 25 de Enero de 2022)

Hammad, M. (2021). Difference between functional and imperative programming. Referencia: https://www.geeksforgeeks.org/difference-between-functional-and-imperative-programming/ (fecha de consulta: 13 de Mayo de 2022)

Hunt, S. (2022). What are virtual machines? Referencia: https://medium.com/@principledminds/virtual-machines-explained-5578371195f (fecha de consulta: 26 de Febrero de 2022)

Inria. (2022). Coq standard library. Referencia: https://coq.inria.fr/library/ (fecha de consulta: 16 de Enero de 2022)

Leroy, X. (2011). Proving a compiler. Referencia: https://xavierleroy.org/courses/Eugene-2011/ (fecha de consulta: 26 de Enero de 2022)

Mack. (2022). Reflexive transitive closure. Referencia: https://math.stackexchange.com/questions/238920/reflexive-transitive-closure (fecha de consulta: 4 de Marzo de 2022)

Oracle. (2022). Java virtual machine specification. Referencia: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.idiv (fecha de consulta: 11 de Abril de 2022)

Pierce, B. C. (2021). Software foundations. Referencia: https://softwarefoundations.cis.upenn.edu/lf-current/toc.html (fecha de consulta: 18 de Octubre de 2021)

Wheeler, D. A. (2022). Sloccount. Referencia: https://dwheeler.com/sloccount/ (fecha de consulta: 12 de Mayo de 2022)

Wikipedia. (2022a). Common languaje runtime. Referencia: https://en.wikipedia.org/wiki/Common_Language_Runtime (fecha de consulta: 28 de Mayo de 2022)

Wikipedia. (2022b). Compiler. Referencia: https://en.wikipedia.org/wiki/Compiler (fecha de consulta: 24 de Abril de 2022)

Wikipedia. (2022c). Liveness-variable analysis. Referencia: https://en.wikipedia.org/wiki/Live-variable_analysis (fecha de consulta: 26 de Mayo de 2022)

Wikipedia. (2022d). Modelo cocomo. Referencia: https://en.wikipedia.org/wiki/COCOMO (fecha de consulta: 10 de Mayo de 2022)

Wikipedia. (2022e). Register allocation. Referencia: https://en.wikipedia.org/wiki/Register_allocation (fecha de consulta: 20 de Marzo de 2022)

Wikipedia. (2022f). Virtual machine. Referencia: https://en.wikipedia.org/wiki/Virtual_machine (fecha de consulta: 4 de Abril de 2022)

Woods, D. (2022). Latex commands. Referencia: https://www.scss.tcd.ie/~dwoods/1617/CS1LL2/HT/wk1/commands.pdf (fecha de consulta: 20 de Marzo de 2022)