Diseño e implementación de un compilador

Alejandro Gadea y Emmanuel Gunther

September 11, 2015

Abstract

En este trabajo describimos la implementación de un compilador para un lenguaje imperativo con clases, abordando las distintas etapas que intervienen en el desarrollo: análisis léxico, parsing, análisis estático, generación de código intermedio y por último código de máquina.
La implementación fue realizada en Haskell, siguiendo una modularización que pretende respetar la división en etapas del desarrollo de compiladores.

1 Descripción del lenguaje

El lenguaje para el cual se ha implementado el compilador es `COMPI`, un lenguaje imperativo simple con definición de clases. Un ejemplo de código es el siguiente:

```
class Program {
  int inc(int x) {
    return x + 1;
  }

  int read_int() extern;

  void print(string s) extern;

  void main() {
    int y;
    y = read_int();
    y = inc(y);
    if (y == 1)
      printf("y==1\n");
    else
      printf("y!=1\n");
  }
}
```

Un programa en `COMPI` consiste de una secuencia de declaraciones de **clases**. Cada clase define atributos y métodos. La sintaxis es similar a la de `C` o `C++`.

El lenguaje es tipado y cuenta con los tipos básicos `int`, `boolean` y `float`. Los strings son un caso particular, ya que no se pueden definir variables string, sino que solo se pueden usar constantes, y puede llamarse a funciones externas que tomen como argumento strings (como en el ejemplo).

Existe el tipo `void` el cual solo se puede utilizar como retorno de métodos (correspondería a la definición de procedimientos). El lenguaje cuenta también con arrays, cuya sintaxis y semántica es similar a `C`.

Con respecto a los arrays, en el texto de referencia del lenguaje hay una pequeña limitación, ya que uno podría definir un array `a` de tipo `A`, donde `A` sea una clase con un atributo definido `attr`, pero sin embargo la gramática no permite escribir `a[0].attr`, lo cual sería algo esperable.

Los atributos definidos en una clase son variables globales a todos los métodos definidos en la misma. Dentro de la definición de un método, se pueden definir bloques, los cuales pueden definir variables locales. Nombres de identificadores iguales sólo pueden ocurrir en distintos niveles de anidado. Por ejemplo, una variable local puede tener el mismo nombre que un atributo de clase, en cuyo caso éste quedará inaccesible hasta la terminación del bloque en cuestión.

Una descripción más extensa del lenguaje se encuentra en el documento de especificación [2].

2 Implementación del compilador

En este trabajo implementamos un compilador que cumple con los requisitos presentados en el documento de especificación de `COMPI`, salvo algunas limitaciones:

- A nivel de **parsing**, se acepta cualquier programa definido según la gramática de `COMPI`. La limitación que mencionamos previamente sobre los arrays fue implementada correctamente (es decir, se puede acceder a un atributo de un índice de un array con tipo `A`, donde `A` es una clase).
- En el **chequeo estático** se realiza el chequeo de tipos y otros chequeos, tal como están descriptas en la sección 3.7 de la especificación.
- Para generar **código intermedio** sólo se permiten programas con una sola clase, y las variables sólo pueden tener tipos básicos. Las constantes de tipo String son permitidas, y se pueden usar para llamar a métodos externos.
- En la generación de código **Assembly**, se implementó toda la traducción de código intermedio, menos para las expresiones de tipo Float.

La implementación fue realizada en el lenguaje **Haskell**.

2.1 Diseño de módulos

El código se dividió en módulos de acuerdo a las etapas de desarrollo de compiladores. Los módulos principales son los siguientes, los cuales se corresponden con directorios en la carpeta raíz del código:

1. **Syntax**. Define el árbol sintáctico abstracto para representar los programas `COMPI`.
2. **Parser**. Define las etapas de análisis léxico y parsing.

3. **StaticCheck**. Define la etapa de chequeo estático. Este incluye **TypeChecker** y el chequeo de otras condiciones estáticas, definidas en la sección 3.7 de la especificación. Estos últimos chequeos están en **Generic**.

4. **InterCode**. Define la traducción de un programa chequeado estáticamente (es decir, no tiene errores de tipos ni los demás errores semánticos chequeados) a un código intermedio, más próximo al código de máquina.

5. **Machine**. Define la traducción del código intermedio a Assembly $x86\_64$.

6. **Main**. Define la función principal para compilar código `COMPI`, implementando todas las opciones que fueron especificadas.

Por cada uno de los módulos anteriores tenemos definido un archivo en el directorio raíz, donde se exportan las funciones que serán visibles desde los demás módulos.

2.2 Sintaxis

El módulo **Syntax** define el árbol sintáctico abstracto (AST) del lenguaje. Está subdivido en los módulos **Syntax.Expr**, **Syntax.Statement**, **Syntax.Program**, **Syntax.Type** y **Syntax.PPrint**. Cada uno de éstos define los tipos de datos que representarán a la sintaxis del lenguaje, de acuerdo a la gramática especificada en [2]. También definimos un pretty printing para poder visualizar un programa parseado.

Todos los tipos definidos para representar el árbol sintáctico son funtores, de manera que podemos agregar información en cada nodo del árbol cuando lo necesitemos. Por ejemplo, podríamos tener información de la línea donde estaba definida una expresión en el archivo de texto, de manera de mostrar información en chequeos posteriores, o podríamos llevar el tipo de la expresión luego del type checker, etc:

```
data Expr' a = ...
```

A estos funtores los llamamos con el nombre correspondiente a la parte de la sintaxis que representan, con el caracter `'` al final del mismo. El parámetro `a` será luego reemplazado por algún tipo concreto dependiendo de la etapa del compilador en la que estemos.

2.3 Lexer y Parser

El módulo **Parser** define el análisis léxico y parsing. La implementación fue realizada utilizando la librería **Parsec** ([1]). Mediante la misma podemos construir parsers utilizando combinadores de alto orden.

En **Parser.Lexer** se definen los tokens para el análisis léxico, usando el tipo `TokenParser` definido en dicha librería. También definimos el tipo del parser: `ParsecL`.

Las funciones para reconocer cada una de las expresiones, sentencias y declaraciones del lenguaje están divididas en tres módulos: **Parser.Expr.**, **Parser.Statement** y **Parser.Program**.

La función principal de parsing es `parseFromString` y está definida en el módulo **Parser**. Toma como entrada el texto de un programa y si no hubo

errores obtiene un árbol sintáctico abstracto, de acuerdo a la definición en **Syntax**, donde el parámetro `a` en los funtores `Expr'`, `Statement'`, `Program'` es reemplazado por el tipo `NoInfo`. Con este tipo no agregamos ninguna información al árbol sintáctico, sin embargo podríamos haber llevado por ejemplo, el número de línea donde fue parseado cada elemento sintáctico.

2.4 Análisis estático

El análisis estático está implementado en el módulo **StaticCheck**, que a su vez se divide en dos grandes módulos: **TypeChecker** y **Generic**.

En **StaticCheck.TypeChecker** se define el chequeo de tipos y alcance de los identificadores. En **StaticCheck.TypeChecker.TypedSyntax** se define el tipo de las expresiones, sentencias y programas bien tipados, que serán el resultado de un chequeo de tipos exitoso.

Cada uno de los submódulos contienen las definiciones para chequear alcance y tipos de los elementos del árbol sintáctico resultante del parsing: expresiones, sentencias, clases y programas. Mediante un transformador de mónada `RWST` ([3]) se realizan estos chequeos, permitiendo tener una mónada **reader** para chequear las declaraciones locales y un **estado** para las globales. El resultado estará encapsulado en una mónada **either** ([4]) pudiendo especificar el error que causa el fallo de esta etapa, en caso de ocurrir.

En **StaticCkeck.Generic** se define el resto de los chequeos estáticos especificados en la sección 3.7 de [2]. Aquí se utiliza una mónada `RWS` y los errores se van acumulando en una lista. En caso que no haya errores la misma será vacía.

Las funciones principales del análisis estático son `typecheckProgram` y `genericChecks`, definidas en **StaticCheck**.

2.5 Código intermedio

En **InterCode** definimos la etapa de generación de código intermedio. Éste consiste de un lenguaje de más bajo nivel, el cual tiene registros infinitos, que se corresponden con las variables originales del programa, los parámetros de las funciones y otros auxiliares necesarios para el cómputo de las expresiones. Por cada uno de éstos (variables, parámetros o auxiliares) tendremos un registro distinto.

Como mencionamos previamente, esta etapa sólo está implementada para programas con una sola clase, y las variables sólo pueden ser de tipos básicos. Un programa en código intermedio consiste de una secuencia de instrucciones, cada una posiblemente precedida por un label.

Como ejemplo, consideremos el siguiente programa `COMPI`:

```
class A {

  int a;
  boolean b;

  int inc(int x) {
    return (x + 1);
  }
```

```
  void printf(string s) extern;

  void main() {

    int i;

    i = inc(2);

    printf("termine");
  }
}
```

El código intermedio generado es el siguiente:

```
                StoreG [(IntType,1),(BoolType,1)]
inc:            Store 2 [(IntType,1),(IntType,1)]
                PopParams [2]
                BAssign (Arith Plus) (R 2) (C (IntL 1)) 3
                PutReturn 3
                ICReturn
main:           Store 4 [(IntType,1)]
                Call inc [C (IntL 2)]
                PopReturn 4
                CallE printf [C (StringL "termine")]
                ICReturn
```

La instrucción `StoreG` va acompañada de una lista que tiene en cada lugar el tipo y el tamaño (por si fueran arrays) de cada variable global, según el orden en que fueron definidas. Los registros del código intermedio destinados a variables globales comienzan en el 0 y el tamaño de dicha lista determina la cantidad de los mismos. En este caso tenemos dos registros globales: el 0 de tipo `IntType` y tamaño 1, y el registro 1, de tipo `BoolType` y tamaño 1.

Para cada definición de función tenemos un label correspondiente al nombre. Luego definimos registros que se corresponden con los parámetros, variables locales y otros auxiliares necesarios para computar el valor de expresiones. En el caso de la función `inc` tenemos un registro correspondiente al parámetro `x` y otro registro auxiliar, necesario para guardar el valor de `(x + 1)`. Estos registros son el 2 y el 3. La instrucción `Store` crea dichos registros con los tipos y tamaños correspondientes.

La instrucción `PopParams` determina cuáles registros de una función son los que corresponden a parámetros (luego en Assembly recuperaremos los valores desde registros del procesador y los moveremos a posiciones de memoria). En este caso el registro 2 es un parámetro de la función.

Para calcular el valor de las operaciones sobre expresiones tenemos instrucciones de asignación, `BAssign` corresponde a operadores binarios, `UAssign` a unarios. Estas asignaciones tendrán siempre un registro auxiliar como destino. En el ejemplo, el registro 3 es el destino de la operación consistente de sumar al registro 2 la constante entera 1.

Al finalizar la función `inc`, utilizamos la instrucción `PutReturn`, que toma como parámetro qué registro es el que debe retornarse. Y luego `ICReturn`, que se corresponde con el retorno de la función.

Lo que sigue en el ejemplo es la función `main`, donde tenemos las instrucciones `Call` y `CallE`, la primera es para llamar a funciones locales y la última para funciones definidas como externas. Esta distinción es necesaria para la traducción posterior a Assembly. Otra instrucción más que aparece en el ejemplo es `PopReturn`, que recupera el valor de retorno de la función que se invoca y lo asigna al registro que toma como parámetro (en este caso el 4).

En **InterCode.InterCode** está definido el tipo que representa las instrucciones de código intermedio.

La traducción del código fuente (bien tipado y sin los errores que se chequean en el análisis estático) al código intermedio está implementada en los módulos **InterCode.Expr**, **InterCode.Statement** y **InterCode.Program**. De manera similar al chequeo estático, aquí también utilizamos una mónada `RWS` para contar con un estado (utilizado para generar labels y la creación de registros) y una mónada reader (utilizada para la asignación de nombres de variables a registros, locales a cada función, como así también para saber a qué label se debe saltar cuando ocurre un *break* o un *continue*, locales a los ciclos).

La función principal de todo el módulo es `generateInterCode`, que se encuentra en **InterCode**, toma un programa bien tipado (`TypedProgram`) y retorna el código intermedio generado.

2.6 Generación de código de máquina

La última etapa del desarrollo del compilador es la generación de código Assembly $x86\_64$. Está implementada en el módulo **Machine**.

La traducción de código intermedio a Assembly la realizamos utilizando un tipo de datos `Instruction` para abstraer las instrucciones que se utilizan al compilar (un subconjunto del lenguaje Assembly es embebido en Haskell utilizando *deep embedding*, en lugar de traducir a string directamente). Luego un pretty printer genera el texto del código final compilado.

Los tipos necesarios para embeber el código Assembly están definidos en **Machine.Code**. El pretty printing está definido en **Machine.PPrint**.

Para la traducción, las instrucciones de código intermedio se traducen de manera bastante directa al código Assembly. Los registros del primero son mapeados a posiciones de memoria relativas a la función donde están definidos.

Como ejemplo, del programa en código intermedio de la sección anterior, se genera el siguiente código:

```
.LC0:
          .string "termine"
          .text

.globl main
.type main, @function

main:           enter    $5, $0
                leaq    -4(%rbp), %rdi
                call    .main
```

```
                leave
                ret
inc:            enter    $16, $0
                movq    %rdi, -8(%rbp)
                movq    %rdi, -8(%rbp)
                movl    %esi, -12(%rbp)
                movl    -12(%rbp), %eax
                addl    $1, %eax
                movl    %eax, -16(%rbp)
                movl    -16(%rbp), %eax
                leave
                ret
.main:          enter    $12, $0
                movq    %rdi, -8(%rbp)
                movq    -8(%rbp), %rdi
                movl    $2, %esi
                call    inc
                movl    %eax, -12(%rbp)
                movl    $.LC0, %edi
                movq    $0, %rax
                call    printf
                leave
                ret
```

Lo primero que tendremos en los archivos generados es un encabezado consistente de todas las constantes string que se encuentren en el código intermedio, y la directiva `.globl` para indicar qué símbolo es accesible desde fuera del programa al momento del link.

Para almacenar los atributos de clase e invocar a la función main de la misma, necesitaremos ejecutar un código previo al correspondiente a las funciones del programa original. Esto lo hacemos definiendo un símbolo `main` y renombramos el label correspondiente al main del programa original para que no haya problemas. La razón para realizar este cambio es que usaremos el compilador gcc para traducir de Assembly a código ejecutable, y en el mismo se busca el símbolo **main** como punto de inicio de la ejecución.

En la función de inicialización `main`, entonces tenemos la instrucción `enter`, que indica que vamos a reservar 5 bytes para variables globales (atributos de la clase).

Cada vez que se llama a una función, se pasará **por referencia** la dirección a la primera variable global. En el ejemplo, corresponde a la variable de tipo entero `a` (las variables se ubican en la parte inferior al *base pointer*), la misma se encontrará en la dirección relativa `-4(%rbp)`, y la siguiente será la variable booleana `b`, ubicada en `-5(%rbp)`.

Luego de `main` tenemos el código de las dos funciones `inc` y `.main`. Cada instrucción en el código intermedio es traducida a una o varias de Assembly. Por ejemplo la instrucción `Store` es traducida a `enter`, y los nombres de los registros son mapeados a las direcciones de memoria donde se ubicarán los mismos dentro de la función. En el caso de `inc` tenemos 16 bytes reservados, correspondientes a la dirección de las variables globales (8 bytes) y a los registros 2 y 3 del código

intermedio de tipo entero (4 bytes cada uno).

En las llamadas a funciones seguimos la convención de pasar los primeros seis parámetros en los registros $\%rdi$, $\%rsi$, $\%rdx$, $\%rcx$, $\%r8$ y $\%r9$. El resto de los parámetros se apilan directamente en el stack.

La implementación de la traducción de todas las instrucciones se encuentra en **Machine.GenCode**. Utilizamos una mónada de estado, donde llevamos la información que irá en el *header* (principalmente las constantes string, cada vez que se encuentra un literal, se agrega al estado el string que luego deberá ser definido en el encabezado. Para generar los labels de estas constantes llevamos en el estado un número natural indicando el máximo utilizado), la información de las variables globales (una lista donde en cada lugar tenemos el tipo y tamaño de cada una, luego si un registro del código intermedio tiene un valor menor a la longitud de dicha lista, significa que corresponde a una variable global, y entonces debemos obtener su dirección en la referencia a la misma) y el mapeo de registros de código intermedio a posiciones de memoria.

Al igual que en el resto de los módulos, tenemos una función principal `generateAssembly`, definida en **Machine**, que toma un programa en código intermedio y retorna el código Assembly junto con el header. Luego se utiliza el **pretty printing** para obtener el texto final.

2.7 Main

Finalmente tenemos el módulo **Main**, donde se define la función principal del compilador para ser ejecutada desde una terminal, teniendo distintas opciones de acuerdo a las etapas que se quieran ejecutar. Se utilizó el módulo `System.Console.GetOpt`, el cual brinda facilidades para parsear opciones en un intérprete de comandos.

Si se ejecuta `$ compi -i Ejemplo.compi`, se obtendrán cuatro archivos de texto cuando no hayan ocurrido errores: `Ejemplo.sint`, `Ejemplo.tysint`, `Ejemplo.ic` y `Ejemplo.s`. El primero corresponde a la salida de la etapa de parsing, se imprime el AST parseado utilizando el pretty printing definido. El segundo corresponde a la salida de la etapa del análisis estático, donde obtendremos el mismo AST pero con la información de los tipos. El tercero corresponde al código intermedio, y el último al código Assembly. También se generará el ejecutable, compilando el código generado con **gcc**.

3 Ejemplos

En el directorio **Ejemplos** incluimos unos archivos `COMPI` para mostrar la funcionalidad del compilador. Cada uno se definió con el objetivo de mostrar las distintas características implementadas:

- `Ejemplo1.compi`. Este programa no se podrá compilar hasta código ejecutable, ya que tiene definidas dos clases, y esa funcionalidad no la incluimos, como dijimos previamente. Sin embargo la idea de este ejemplo es ver el funcionamiento de los módulos de parsing y chequeos estáticos. Para correrlo se debe ejecutar `compi` de la siguiente manera:

```
compi -i Ejemplo1.compi -t staticcheck
```

y se obtendrán los archivos correspondientes a las dos primeras etapas. En estos archivos se puede ver el árbol sintáctico obtenido, con la información de los tipos en el segundo caso. Vale aclarar que el modo en que se imprimen estos archivos no es el más claro posible, debería hacerse un pretty printing del AST más ameno.

Para ver algunos de los errores que se obtienen en estas etapas, se puede modificar el ejemplo de manera que ocurran éstos. Por ejemplo, si cambiamos en la línea 14 (función `get1`), la variable `attr1` por `attr2` obtendremos el siguiente error:

```
$ compi -i Ejemplo1.compi -t staticcheck

Type Checker Error:

El tipo "boolean" no coincide con el tipo "int"
en la expresión (attr2) : boolean
```

- `Ejemplo2.compi`. En este programa testeamos el alcance de los identificadores. Para ello definimos un atributo con nombre `i`, y variables locales con el mismo nombre en distintos bloques. Cambiamos los valores de cada uno e imprimimos el valor para ver el correcto funcionamiento.

  Para correr este ejemplo ejecutamos `compi` sólo indicando el archivo de entrada:

  ```
  compi -i Ejemplo2.compi
  ```

  Obtendremos los cuatro archivos de texto correspondientes a las etapas de compilación y un ejecutable.

- `selectStort.compi` Por último definimos un ejemplo más completo y con una funcionalidad más interesante. Buscamos en internet código C con el algoritmo *selection sort*. Lo modificamos un poco para adaptarlo a `compi` (lo cual fue casi directo, acomodando las funciones dentro de una clase). En el mismo utilizamos dos funciones externas definidas en la librería estándar de C: `printf` y `rand`, para imprimir el resultado y calcular números aleatorios con los cuales inicializar un arreglo que luego será ordenado.

  La ejecución del programa produce la siguiente salida:

  ```
  $ ./selectSort
  Array to sort:
  383
  886
  777
  915
  793
  335
  386
  ```

```
492
649
421
Sorted array in ascending order:
335
383
386
421
492
649
777
793
886
915
```

4 Conclusión

La implementación de un compilador más o menos realista en todas sus etapas fue un trabajo muy interesante y brindó la posibilidad de aprender mucho, no solo de compiladores en general, sino también sobre el desarrollo modular de un software de tamaño mediano.

A medida que avanzamos con el desarrollo nos fuimos encontrando con las dificultades propias de la implementación de un compilador: Determinar los módulos que compondrán cada etapa, elegir un código intermedio adecuado, comprender el lenguaje ensamblador de acuerdo a la arquitectura requerida, etc. En muchas ocasiones hubo que volver a módulos anteriores para realizar algún cambio que surgió a posteriori. El caso más claro de esto fue en la definición del código intermedio, donde al traducir a Assembly se hacía más evidente cómo debía ser ese lenguaje.

Un punto a destacar es la elección del lenguaje de implementación. Utilizar **Haskell** fue un gran acierto ya que permite manipular expresiones y diseñar lenguajes de una manera muy prolija, mediante el uso del pattern matching, como así también las ventajas de separar código *puro* de aquel con efectos secundarios. Para esto contar con mónadas fue muy útil, en particular la mónada *reader* posibilitó implementar el anidamiento en bloques del lenguaje `COMPI` de una manera transparente.

En conclusión, si bien a medida que se avanzó en las etapas se introdujeron algunas limitaciones para simplificar el trabajo, se pudo comprender y experimentar todo lo que conlleva desarrollar un compilador para un lenguaje más o menos realista.

References

[1] Paolo Martini Daan Leijen. Parsec parsing library. `https://hackage.haskell.org/package/parsec`, 2006.

[2] Marcelo Arroyo Francisco Bavera. Descripción de compi. `http://dc.exa.unrc.edu.ar/moodle/pluginfile.php/6977/mod_folder/content/0/01-COMPI-spec-lenguaje.pdf`, 2015.

[3] Andy Gill. Monad classes, using functional dependencies. `http://hackage.haskell.org/package/mtl-2.2.1`, 2001.

[4] Edward A. Kmett. An either monad transformer. `http://hackage.haskell.org/package/either`, 2011.