Técnicas de Compilación y Optimizaciones para el Lenguaje Occam

Guillermo A. Alvarez * Daniel O. Bandinelli *

Marcelo O. Fernández * Gustavo A. Rosini *

Resumen

Occam es un lenguaje de programación basado en un modelo de procesos secuenciales que se ejecutan concurrentemente, comunicándose por medio de pasaje sincrónico de mensajes. Se detallan los aspectos del desarrollo de un compilador Occam–C sobre una máquina uniprocesador. El sistema operativo UNIX es usado como plataforma de implementación, empleando las facilidades que brinda para programación concurrente y comunicación entre procesos. Se incluyen explicaciones detalladas de los chequeos realizados, de los procedimientos de optimización, y de las técnicas de compilación desarrolladas. Los chequeos permiten asegurar la validez del programa de entrada. Los procedimientos de optimización ideados, aplicados tanto durante la generación de código como durante su ejecución, permiten la utilización eficiente de los recursos del sistema por parte del código objeto. Las técnicas de compilación definen la traducción de un lenguaje diseñado para arquitecturas multiprocesador a sistemas monoprocesador time-sharing. Si bien algunos de los algoritmos utilizados para estos fines son estándares en construcción de compiladores, se enfatiza la descripción de las ideas originales desarrolladas para este proyecto. Estas se basan en las propiedades del lenguaje Occam, y de la arquitectura sobre la cual se ejecuta el código generado.

Palabras clave: Compiladores, Lenguajes de Programación, Programación Concurrente, Sistemas Operativos.

1 Introducción

Durante las últimas dos décadas, diferentes paradigmas de programación paralela han sido propuestos. Entre ellos se encuentran los paradigmas basados en pasaje de mensajes como Occam [INMOS 84]. Esta familia de lenguajes tiene su sustento teórico en CSP [Hoare 85], y admite como soporte a arquitecturas multiprocesador con memoria distribuida y pasaje de mensajes.

Este trabajo describe el desarrollo de un compilador para el lenguaje concurrente Occam que genera código C sobre el sistema operativo UNIX System V. Si bien el lenguaje Occam fue

*Escuela Superior Latino Americana de Informática (ESLAI) - CC 3193 - (1000) Buenos Aires - Argentina; E-mail: alvarezg@buevm2.vnet.ibm.com, danielb@dcfcen.edu.ar

concebido originalmente para ser ejecutado en un array de procesadores homogéneos (Transputers), en nuestro caso se simula la ejecución paralela de procesos secuenciales utilizando las facilidades provistas por UNIX para programación concurrente y comunicación entre procesos.

El compilador acepta un subconjunto del lenguaje Occam standard con la adición de extensiones destinadas a facilitar la interacción con el usuario. Dichas extensiones permiten entrada/salida respetando el modelo conceptual del lenguaje.

La compilación de un programa consiste en dos etapas que se ralizan secuencialmente. La primera (*front-end*) acepta como entrada el texto de un programa Occam sintácticamente correcto que cumple las restricciones inherentes al lenguaje, y lo transforma en código para una máquina abstracta. La segunda (*back-end*) toma la salida de la primera y genera código para una máquina real en términos del código para la máquina abstracta. Cada una de estas etapas se efectúa en una pasada sobre el código, y por lo tanto el proceso total de compilación insume solamente dos pasadas.

El trabajo está organizado como sigue. En la Sección 2 se describen las generalidades del lenguaje Occam. Las Secciones 3 y 4 detallan aspectos del front-end y del back-end, respectivamente. En la Sección 5 se explican las optimizaciones implementadas; por último, en la Sección 6 se presentan conclusiones y futuros trabajos.

2 Generalidades del Lenguaje Occam

Occam es un lenguaje concurrente, basado en el CSP de Hoare. Está diseñado para soportar aplicaciones concurrentes en las cuales muchas partes de un sistema operan independientemente e interactúan. Un programa se expresa en términos de procesos concurrentes que se comunican enviando y recibiendo mensajes a través de canales de comunicación. Occam provee comunicación sincrónica entre procesos que se ejecutan en paralelo, selección no determinística y creación dinámica de procesos secuenciales. Además provee facilidades para programación en tiempo real, a través de constructores que permiten un manejo preciso del tiempo con prestaciones de timeout y alarmas. En los siguientes párrafos se describen las construcciones principales del lenguaje, recurriendo a ejemplos cuando sea necesario.

Procesos Primitivos: Estos procesos consisten en una acción elemental, y pueden combinarse para formar procesos arbitrariamente complejos. Los procesos primitivos consisten en asignación, entrada y salida de canales y bloqueo durante un período determinado de tiempo. El proceso `var := exp` asigna a la variable `var` el valor de la expresión `exp` y termina; los procesos `canal ? var` y `canal ! exp` leen y escriben respectivamente un valor por un canal de comunicación entre procesos, terminando solamente cuando la operación se haya completado (es decir que tanto la lectura como la escritura son *sincrónicas*). Como extensión al lenguaje Occam estándar, se definieron e implementaron extensiones que soportan entrada/salida con el usuario en un ambiente UNIX. Dichas extensiones toman la forma de dos canales predefinidos (`IN` y `OUT`) cuya función es comunicar a los procesos que los usen con el `stdin` y `stdout` de UNIX respectivamente. Por otra parte, el proceso `WAIT NOW AFTER exp` termina cuando el valor del reloj local sobrepasa el valor de la expresion `exp`. Además se provee un proceso `SKIP`

```
CHAN comun_interna:
  PAR
    WHILE TRUE
      VAR x:
      SEQ
        can1 ?  x
        comun_interna !  x*x
    WHILE TRUE
      VAR y:
      SEQ
        comun_interna ?  y
        can2 !  y*y
```
(a)

```
WHILE TRUE
  VAR x:
  ALT
    cin1 ?  x
      cout !  x
    cin2 ?  x
      cout !  x
```
(b)

Figura 1: Programas Ejemplo: (a) Constructor `PAR`; (b) Constructor `ALT`.

que no realiza ninguna acción y termina sin ningún efecto.

Constructores de Procesos: Dichos constructores crean nuevos procesos a partir de procesos menos complejos. Las opciones que Occam provee se detallan a continuación.

Los constructores `IF THEN, IF THEN ELSE, WHILE` son similares a los presentes en los lenguajes usuales de programación secuencial. El constructor `SEQ` toma un conjunto de procesos y un orden, y consiste en el proceso en el cual cada uno de sus componentes se ejecuta secuencialmente en el orden preestablecido.

El constructor `PAR` toma un conjunto de procesos, y los ejecuta concurrentemente; termina cuando todos sus componentes han finalizado sus ejecuciones. Como un ejemplo de su uso, la Figura 1 muestra un programa que calcula x^4 leyendo el valor de x del canal `can1` y devolviendo el resultado por el canal `can2`.

El constructor `ALT` provee una selección alternativa entre un grupo de procesos. Cada uno de dichos procesos tiene asociada una *guarda*. Una guarda Occam puede estar formada por un comando `WAIT`, o una lectura de canal, o un comando `SKIP`, posiblemente precedidos por una expresión booleana. Las guardas son verdaderas cuando la expresión booleana (si existe) es cierta y el proceso elemental asociado completa su ejecución. Por ejemplo, una guarda de la forma `exp & canal ? var` será verdadera cuando `exp` sea cierta y la operación de lectura haya resultado exitosa, o sea cuando otro proceso haya realizado una escritura en el otro extremo del canal. En el caso de que dos guardas se hagan verdaderas simultáneamente se elige una de ellas en forma no determinística. La Figura 1 muestra un ejemplo del uso del constructor `ALT`, donde los datos provenientes de los canales `cin1` y `cin2` son enviados por el canal `cout` a medida que se van recibiendo.

Expresiones: Se proveen los operadores aritméticos de suma, diferencia, multiplicación y división, junto con los operadores de comparación, booleanos y lógicos (bitwise).

Pueden definirse constantes enteras, y existen dos constantes predefinidas `TRUE` y `FALSE`. La primera tiene como valor una palabra con todos sus bits en 1, y la segunda una palabra con todos los bits en 0.

En la definición informal de la semántica de Occam [INMOS 84] no está clara la forma en que se debe operar con estos datos dentro de las expresiones. Una expresión es *entera* si su operador más externo es aritmético o lógico; es *booleana* en caso contrario. Se garantiza para las expresiones booleanas que si son verdaderas su valor es la constante `TRUE`, y que si son falsas su valor es la constante `FALSE`. Las expresiones que aparecen como condición en las construcciones `IF`,`WHILE`,`WAIT` y `ALT` pueden ser tanto booleanas como enteras, y en el caso de las últimas se adopta la convención de que un valor distinto de cero es verdadero y un valor cero es falso.

Para poder cumplir con el requerimiento antes citado de que la evaluación de una expresión booleana arroje como resultado una de las constantes booleanas `TRUE` o `FALSE`, es necesario que el compilador realize coerciones. Las coerciones serán usadas sólo cuando sea imprescindible; por ejemplo, dada una expresión booleana sumada a otro número, sólo es coercionada cuando se distingue que será usada como argumento de la suma, porque en otro caso podría haber sido utilizada como condición en un `IF`, y en este caso hubiera sido superfluo coercionarla. En general, las coerciones se harán sobre expresiones booleanas en tres casos:

1. Cuando la expresión sea argumento de cualquier operador entero
2. Cuando la expresión es la parte derecha de una asignación
3. Cuando la expresión es enviada por un canal

En cuanto al orden de evaluación de las expresiones booleanas, se garantiza que la evaluación se abandona tan pronto como sea posible, comenzando de izquierda a derecha (*short circuit*).

3 Chequeos y Generación de Código Intermedio

En esta sección se detallarán los diversos chequeos realizados sobre el programa Occam de entrada. Para realizar dichos chequeos, se debe recoger durante la fase de análisis del texto del programa cierta información relativa a los identificadores. La información que debe guardarse para un identificador Occam es básicamente de tres clases:

1. Atributos intrínsecos: { Nombre; Tipo (variable, canal o constante); Valor (para las constantes) }
2. Información de scope
3. Información de posición: Denota qué tipo de accesos al identificador se hacen y en qué lugares (procesos) del texto fuente se los hace. Es usada primordialmente para garantizar que la comunicación entre procesos se realize utilizando canales y no variables compartidas. Además sirve para chequear la correcta utilización de los canales.

Por intermedio de los chequeos realizados hemos podido determinar tanto errores fatales como warnings que ayudan al programador a realizar un código más confiable. Los chequeos realizados son los siguientes.

Utilización de variables compartidas: Se dice que dos procesos pertenecientes a un constructor `PAR` *acceden* a una variable compartida si ambos la escriben o uno la escribe y el otro la lee. Esto quiere decir que dada una construcción de la forma

$$\texttt{PAR}\ P_1 \ldots P_n$$

si llamamos $W(P)$ al conjunto de variables escritas por el proceso P y $R(P)$ al conjunto de variables leídas por el proceso P, deben cumplirse las siguientes condiciones (conocidas como *condiciones de Bernstein*)

$$W(P_i) \cap W(P_j) = \emptyset$$

$$W(P_i) \cap R(P_j) = \emptyset$$

$$1 \leq i, j \leq n \ ; \ i \neq j$$

Una solución posible para chequear que dichas condiciones se cumplan es operar con los conjuntos de lectura y escritura de cada uno de los procesos componentes de un constructor `PAR`. Sin embargo, esta alternativa fue rechazada porque implica operaciones costosas entre conjuntos y dificulta la localización exacta de errores.

Debido a esto hemos buscado mejores maneras de diseñar este chequeo. En los párrafos siguientes se presenta un algoritmo original que nos permite detectar el error en el lugar donde se produjo en una forma más eficiente. Para ello observamos que los lugares en donde se puede incurrir en un error son

- entrada y asignación (escritura de la variable)
- expresiones (lectura de la variable)

Los procesos que están involucrados aquí son los básicos, proceso guardado, `IF THEN`, `IF THEN ELSE`, `WAIT` y `WHILE`. Así asignamos un identificador a cada uno de estos procesos de tal forma que podemos determinar, dados dos identificadores, si los procesos correspondientes *pueden ejecutarse en paralelo* sin hacer suposiciones sobre los tiempos de ejecución de cada proceso. Bajo la hipótesis de que en las líneas previamente reconocidas no hay violaciones a las condiciones de Bernstein, el chequeo puede realizarse de la siguiente manera:

1. Si un proceso lee una variable, se toma el identificador de dicho proceso y se verifica que esta lectura no pueda ejecutarse en paralelo con el último proceso (en el orden en que se reconoce) que escribió la variable.
2. Si un proceso escribe una variable, se toma el identificador de dicho proceso y se verifica que esta escritura no pueda ejecutarse en paralelo con el último proceso que escribió la variable y que tampoco pueda ejecutarse en paralelo con cada uno de los procesos que leen esa variable.

El inconveniente es encontrar una manera de identificar a los procesos que nos permita determinar si pueden ejecutarse en paralelo. La manera en que lo hacemos es obligando a que cada identificador de proceso muestre el lugar sintáctico en donde se encuentra el proceso. Para determinar el lugar sintáctico solo nos interesan en este caso los constructores `PAR, ALT, SEQ` que nos dicen cómo se ejecuta un proceso con respecto a otro. Un identificador es un string de elementos del conjunto $\{$ `P`, `A`, `S`, `1`, `2`, `3`, $\ldots\}$ y el operador "$\oplus$" concatena un string con un elemento.

La forma en que construimos los identificadores es:

1. Antes de comenzar a compilar el programa el identificador de cada proceso es ε (identificador vacío)
2. Si Id es el identificador de `PAR` $P_1 \ \ldots P_n$ entonces el identificador de P_i es $Id \oplus$ `P` $\oplus\, i$
3. Si Id es el identificador de `ALT` $P_1 \ \ldots P_n$ entonces el identificador de P_i es $Id \oplus$ `A` $\oplus\, i$
4. Si Id es el identificador de `SEQ` $P_1 \ \ldots P_n$ entonces el identificador de P_i es $Id \oplus$ `S` $\oplus\, i$

Así, dos procesos P_1 y P_2 pueden ejecutarse en paralelo si

$$Ult(Pre) = \texttt{P}$$

donde Pre es el prefijo más largo que tienen en común los identificadores de P_1 y P_2, y $Ult(S)$ es el último elemento del string S.

Cuando el chequeo de utilización de variable compartida no se satisfaga se indicará con el mensaje de error correspondiente.

Utilización de canales: Un canal es *correctamente utilizado* si establece una conexión entre dos procesos pertenecientes a un constructor `PAR` en donde sólo uno envía y sólo uno recibe por dicho canal. Esto requiere chequear que:

- Dos lecturas (escrituras) sobre un canal no puedan ejecutarse en paralelo, o sea que sólo un proceso lee (escribe) sobre el canal
- Cada una de las lecturas deban poder ejecutarse en paralelo con las escrituras y viceversa
- Al menos un proceso lea del canal y uno escriba sobre el canal; esto asegura que el canal sea una conexión entre dos procesos

A continuación se presenta el chequeo ideado, que respeta los tres items anteriores

- Para asegurar el primer item sólo se necesita que dada una lectura (escritura) se chequee que dicha operación no pueda ejecutarse en paralelo con la última lectura (escritura)
- El segundo item se cumple si dada una lectura (escritura) verificamos que esta operación se pueda ejecutar en paralelo con la primera escritura (lectura)

- Para garantizar el último item se controla al terminarse el alcance de un conjunto de declaraciones que para cada canal exista al menos una lectura y una escritura sobre él

De acuerdo a esto, por cada canal, sólo es necesario almacenar en la tabla de símbolos quiénes son el primer y último proceso que lee y escribe cada canal. Para individualizar a dichos procesos utilizamos los identificadores que definimos para el chequeo de variables.

Siempre que se detecte una violación a alguno de los tres puntos se indicará el mensaje de error correspondiente.

Constantes, variables y canales no utilizados: Se detectan los identificadores declarados pero no utilizados en el texto Occam, notificándose con el warning correspondiente.

Variable que es leída y no está inicializada: También se avisa (a través de un warning) de esta situación para prevenir al programador de un comportamiento posiblemente no determinístico del programa.

Los identificadores utilizados deben estar declarados una sola vez: Se chequea que todo identificador esté declarado en el scope en que se lo utiliza. Además no permitimos declaraciones de un identificador cuando ya haya sido declarado en el mismo scope con un tipo distinto. La violación de cualquiera de estos dos requisitos resulta en un error.

Uso de un identificador de acuerdo a su tipo: Se verifica que los identificadores se usen consistentemente con sus declaraciones, por ejemplo que en una escritura por canal no se pretenda escribir el dato sobre una variable entera.

Programa que no termina: La evaluación y propagación estática de constantes permiten determinar casos en los que el programa itera infinitamente; se notifica esta situación emitiendo un warning.

La expresión que aparece en un proceso `WAIT` debe ser una comparación de tiempos: Sólo se permiten procesos `WAIT` de la forma `WAIT NOW AFTER exp` donde `exp` no contiene ocurrencias de `NOW`.

Un solo tipo de operador asociativo debe aparecer en una expresión no parentizada: No se permite que una secuencia de expresiones esté operada con operadores asociativos distintos, a menos que esté máximamente parentizada.

Para escribir el front-end se utilizaron el generador de analizadores lexicográficos `lex` y el generador de parsers `yacc` provistos por UNIX. Estos brindan una forma clara y cercana al modelo teórico de especificar el lenguaje que se desea reconocer usando esquemas de traducción [Aho et al. 86]. Además permiten modificar la gramática con poco esfuerzo lo cual no ocurriría si tanto el parser como el analizador lexicográfico estuvieran implementados directamente en

C. El hecho de utilizarlos para el desarrollo del compilador favorece su portabilidad a otros sistemas UNIX.

El código intermedio generado como consecuencia de la primera etapa de la compilación tiene la forma de un árbol sintáctico, que refleja la estructura del programa fuente después de que se le realizan las optimizaciones detalladas en la Sección 5.

4 Generación de Código

La presente sección contiene aspectos relativos a las prestaciones que brinda el soporte de tiempo de ejecución implementado como complemento del compilador, y a la generación de código C a partir de los principales constructores Occam.

4.1 Soporte de Tiempo de Ejecución

La traducción de Occam a C involucra la utilización de facilidades de sincronización y comunicación entre procesos, por lo cual el soporte de tiempo de ejecución permite crear y liberar dinámicamente dichos recursos en forma eficiente. Esto se logra a través de manejadores que se encargan de administrar el uso de canales de comunicación, variables en memoria compartida y semáforos.

El manejador de canales se construyó usando las facilidades provistas por los manejadores de semáforos y de memoria compartida. La comunicación entre procesos es unidireccional, es decir hay un proceso emisor del mensaje (en Occam este mensaje es un entero) y el otro proceso es el receptor, no pudiéndose intercambiar ambos papeles. Además la comunicación es sincrónica, esto implica que se lleva a cabo cuando ambos procesos (el emisor y el receptor) quieren comunicarse. Si sólo uno de los procesos está listo para la comunicación debe esperar a que el otro también lo esté.

De los identificadores Occam, se marcan como residentes en memoria compartida sólo a aquellos que probablemente se usen para comunicar resultados de un proceso hijo a su padre. Esto ocurre cuando un componente de un constructor `PAR` modifica una variable que es leída posteriormente por un proceso que se ejecuta después de que el `PAR` termina. Dado que UNIX no permite crear variables compartidas individuales (porque se debe crear un segmento de memoria compartida cuyo tamaño debe ser mayor que un límite fijado por el sistema operativo [Bach 86, AT&T 84]), se debe implementar un manejador intermedio que permita efectuar pedidos de memoria con mayor frecuencia y granularidad más fina. Como las variables compartidas van a ser usadas por varios procesos simultáneamente, las estructuras utilizadas por el manejador deben estar en memoria compartida para poder ser accedidas por todos los procesos.

Para manejar semáforos se han definido operaciones según la notación usual (definida en [Dijkstra 65]). Se debe tener en cuenta que UNIX [AT&T 84, Stevens 90] no maneja semáforos individuales sino arreglos de los mismos. Este problema es similar al visto anteriormente para administrar las variables compartidas, y se soluciona implementando un manejador que sigue los mismos principios.

Las variables compartidas y los canales del programa Occam determinan la generación de dos clases de código: declaraciones para los identificadores C asociados a ellos, y código

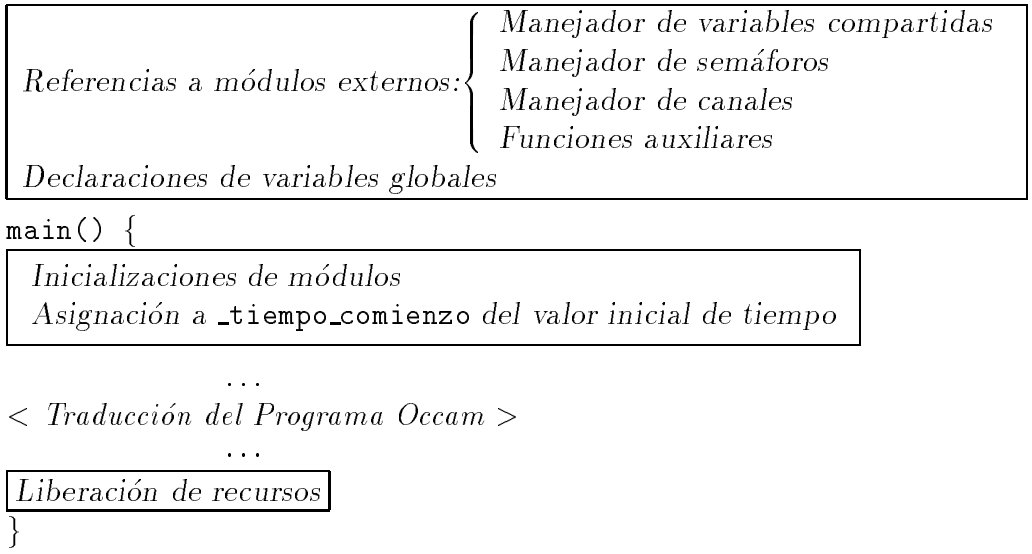

Figura 2: Formato Típico del Código C Generado.

ejecutable C que invoca a los respectivos manejadores, creando y liberando los recursos cuando los correspondientes identificadores entran y salen de scope.

4.2 Estructura del Generador de Código

En los siguientes párrafos se describen las formas en que los principales constructores Occam son traducidos a código C. Las técnicas de compilación concebidas a tal fin hacen hincapié en la eficiencia de ejecución del código generado, disminuyendo la cantidad total de procesos activos, evitando el busy-waiting, y maximizando el paralelismo entre cómputo y entrada/salida del sistema.

Dado un árbol que representa el código intermedio del programa Occam, se lo recorre recursivamente generando el correspondiente código C. Para ello se genera un encabezamiento que produce la inclusión de los headers de los manejadores de semáforos, canales y variables compartidas, y además declara algunas variables globales del soporte de tiempo de ejecución. Luego se genera la declaración de la función `main()` donde se inicializan los manejadores y las variables del soporte, y a continuación el código correspondiente al árbol; se termina con el cierre de los módulos utilizados.

El formato de un programa C típico, generado como salida del back-end, es mostrado en la Figura 2.

Declaraciones :
Al procesar el código intermedio correspondiente a un bloque de declaraciones Occam, se genera un bloque C donde las variables Occam se declaran como variables enteras si no son compartidas. Para los canales, generamos una declaración de tipo `_chan`.

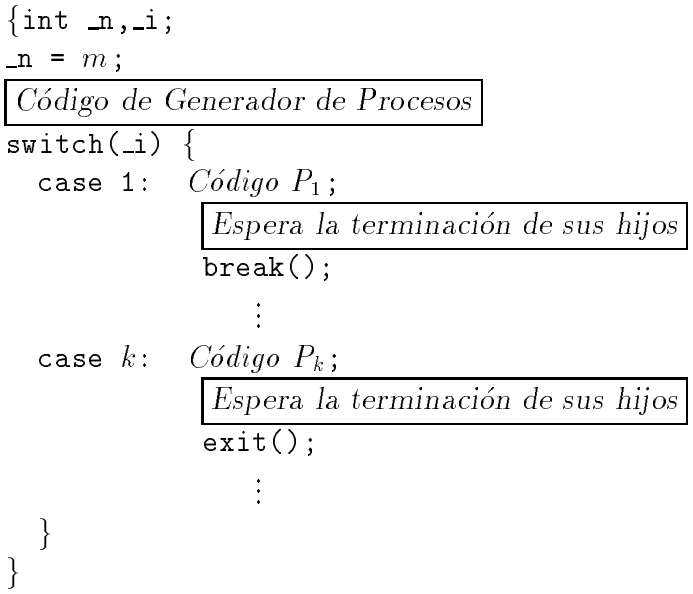

Figura 3: Traducción del constructor `PAR`.

Después de generar las declaraciones del bloque C y antes de comenzar con el código, se generan los `create_vc()` y los `create_chan()` correspondientes, para que los manejadores creen los recursos en tiempo de ejecución. Análogamente, al terminar el procesamiento del bloque, generamos los `remove_vc()` y `remove_chan()` correspondientes antes de cerrar el bloque C.

No es necesario generar declaraciones para las constantes Occam en esta etapa, ya que al construir el árbol las apariciones de las constantes se reemplazaron por el valor correspondiente.

Cabe notar que se generan únicamente las declaraciones correspondientes a aquellas variables y canales que realmente fueron usados, y no para los que fueron declarados pero no utilizados, lográndose de esta manera el consecuente ahorro de espacio de almacenamiento y de overhead de creación en tiempo de ejecución.

Constructor PAR:

La traducción del constructor `PAR` consiste en generar código que crea dinámicamente procesos UNIX, cada uno de los cuales ejecuta uno de los procesos componentes del constructor. La creación de procesos se hace en forma de árbol binario completo por niveles para mejorar la eficiencia, como se explicará en la Sección 5.

Por lo tanto, una construcción Occam de la forma `PAR` P_1 ... P_m se traducirá, si $m > 1$ (si no, se genera código directamente para el único proceso), en el código C de la Figura 3.

La variable `_n` indica al generador la cantidad de procesos a crear; a la salida, `_i` contiene el número de orden de cada proceso generado, por lo cual puede elegir la rama correcta del `switch()` y ejecutar su código. Cada proceso genera a lo sumo dos hijos (según su posición en el árbol), ejecuta su rama del `PAR` y luego espera que sus hijos terminen. El proceso padre ejecuta la primera componente del constructor. Dentro del código del generador se tienen en cuenta posibles errores de ejecución al excederse la cantidad máxima de procesos permitida por el sistema.

Constructor ALT:
Si alguna de las guardas del `ALT` se revela verdadera en tiempo de compilación, entonces se genera solamente el código del proceso guardado correspondiente, como una implementación del no-determinismo.

La política que se adoptará al evaluar las guardas será considerarlas como una generalización del concepto de expresiones booleanas: una guarda (en el caso más complejo) será cierta cuando la expresión booleana que está a la izquierda del `&` sea cierta, y además la operación que está a la derecha pueda realizarse. En este contexto, esta expresión se evaluará siguiendo la semántica estándar de Occam, de izquierda a derecha y con corto circuito. Por lo tanto, al ejecutarse el código generado a partir de un `ALT`, se evaluarán primero todas las expresiones que figuren en las partes izquierdas de guardas, y se descartarán definitivamente las que den `FALSE`. Esta decisión posibilita una implementación en la que ninguno de los casos posibles provoca espera activa, como se verá más adelante, ya que aún en el caso de expresiones que involucren a la función primitiva `NOW` su valor de verdad se evaluará una sola vez.

Tras este paso inicial de evaluación, pueden existir guardas de la forma `exp & SKIP` cuya expresión haya evaluado a `TRUE`, y que por lo tanto sean ciertas. Es por esto que se genera código para verificar al ejecutar si esto ocurre, y en el caso de que ocurra se dice que el `ALT` se ha *trivializado*: se ejecutará el proceso guardado correspondiente.

Las guardas que pasaron la primera etapa (sus expresiones son verdaderas, o bien consisten en una entrada por canal o `WAIT` aislados) son de dos clases distintas, asumiendo que el `ALT` no se ha trivializado: entradas o procesos `WAIT`. De aquí en adelante sólo nos referiremos a estas guardas. Para implementar su evaluación, se genera en tiempo de corrida un proceso para cada una de estas guardas. La generación de procesos se hace en forma plana y no de árbol binario ya que si se mantuviera la estructura de creación de árbol binario, podría pasar que algunos procesos fueran descartados por ser falsas las expresiones de sus guardas, pero éstos no generarían a sus hijos en el orden arbóreo, que sí podrían tener guardas válidas. Dicha generación de procesos se hace sólo en los casos en que el `ALT` no se ha trivializado y no todas las guardas tienen expresiones falsas como componente.

Ninguno de estos procesos hace espera activa, ya que los encargados de evaluar `WAIT`s calculan la cantidad de tiempo que tendrá que pasar y ejecutan un `sleep()` de esa duración; por otra parte, los encargados de evaluar entradas se duermen mientras no haya un dato disponible en el canal. Una implementación alternativa para el `ALT` hubiera sido evaluar continuamente las guardas hasta que alguna se cumpliera, desperdiciando tiempo de CPU en actividades inútiles. Fue necesario definir un mecanismo para que el proceso cuya guarda se cumpla primero avise al proceso padre (coordinador) que debe ejecutar su correspondiente proceso guardado. Para ello, el padre genera primero los procesos evaluadores de guardas y luego se queda suspendido en el semáforo `_alguien_termino` (que inicialmente vale 0). Cuando un proceso hijo puede cumplir su guarda, verifica primero si puede avisarle al padre, para garantizar que el padre aceptará a sólo un proceso; si puede, dejará su número en un espacio de memoria compartida, y luego ejecutará un `V(_alguien_termino)` despertando por lo tanto al padre. El padre, una vez enterado de que todos sus hijos instalaron sus manejadores (a través del semáforo `_activados`), espera hasta que uno de ellos triunfe; sólo entonces interrumpe enviando señales a los demás procesos hijos para que éstos terminen su ejecución. Esto se hace para no esperar a guardas

demasiado lentas, y para reducir la cantidad de procesos en el sistema al mínimo. Un hijo que satisface su guarda ejecuta un `P(_primero_que_termina)` (inicialmente en 1), para que si dos guardas se cumplen al mismo tiempo sólo una pueda avisar al padre y la otra quede suspendida en dicho semáforo.

Los procesos que esperan entradas sobre el mismo canal necesitan que sólo uno de ellos retire un dato de ese canal; para lograr esto, se define un semáforo auxiliar por cada canal (`_sem_`*nomcanal*) inicialmente en 1, que sólo permite que una guarda ejecute la lectura sobre dicho canal.

Los procesos que esperan entradas sobre canales distintos deben mantener la semántica de Occam en el sentido de que sólo el que logra ejecutar su proceso guardado retira el dato del canal, dejando los demás canales sin cambio (en ellos no se retira el dato ni se desbloquea al proceso que lo envió), y además tampoco altera las variables que figuran como destino en las entradas no exitosas. Para solucionar el problema, se define un `receive_guard()` sobre canales, el cual no desbloquea al proceso que envió el dato hasta haberle avisado al padre que él es la guarda verdadera. Y para no alterar variables destino de entradas supuestamente no ejecutadas, se definen variables auxiliares sobre las cuales realmente se hace la entrada. Al comienzo del proceso guardado, se asignan estas variables a la verdadera variable destino. Por lo tanto, la evaluación de guardas a cargo de procesos independientes no causa efectos laterales indeseables.

Cuando un proceso evaluador de guarda es interrumpido por su padre, pasa a ejecutar un manejador, que en el caso de los `WAIT` y de las entradas desde el `stdin` se limita a abortar la ejecución, pero en el caso de las entradas desde canales Occam se fija si estaba a la mitad de la lectura, restaurando en ese caso el estado del canal.

Por lo tanto, una construcción Occam de la forma `ALT` $\begin{matrix} G_1 & \dots & G_m \\ P_1 & & P_m \end{matrix}$ se traducirá en el código C de la Figura 4, donde *nomvar* es la variable sobre la cual se realiza la entrada.

Demás Constructores :

Los constructores `SEQ, IF` y `WHILE` se traducen de la forma natural. Las lecturas y escrituras sobre variables en memoria compartida se realizan invocando las primitivas del manejador correspondiente. Lo mismo vale para la asignación.

En cuanto a las lecturas y escrituras de canales, se realizan a través de las correspondientes primitivas sincrónicas provistas por el manejador de canales, salvo en el caso de los canales `IN` y `OUT`. Para estos casos se generan llamadas a las rutinas estándares de biblioteca de C para entrada/salida.

El constructor `WAIT` se compila como un código que duerme al proceso durante el tiempo resultante de la diferencia entre el valor corriente de `NOW` (es decir, el reloj del sistema) y el momento en el cual la ejecución deberá continuar.

```
{int _i,_n,_proc_guard = 0,_alguna_exp_cierta = 0,_alt_simplificado = 0;
_semaforo _alguien_termino,_primero_que_termina,_activados;
_vc _termino;
Declaraciones de Otras Variables y Semáforos Auxiliares
create_vc(&_termino);
create_sem(&_alguien_termino,0);
create_sem(&_primero_que_termina,1);
Creación e Inicialización de Variables y Semáforos Auxiliares
_n = m ;
Evaluación de las Expresiones Booleanas
if (!_alt_simplificado || _alguna_exp_cierta){
   create_sem(&_activados,m);
   Generador Plano de Procesos
   switch(_i){
       case 1: Código G1
              exit();
                 ⋮
       case m + 1:P(&_alguien_termino);
                  _proc_guard = read_vc(_termino);
                  wait_for_zero(&_activados);
                  Enviar Interrupción a los Hijos no Exitosos
                  Esperar Fin de Hijos
       }
   }
switch(_proc_guard) {
   case 1: Código P1
           break;
            ⋮
   case m: Código Pm
           break;
   }
remove_vc(_termino);
Liberación de Variables y Semáforos
}
```

(a)

```
signal(ABORT,_handler_wait);
P(&_activados);
Código de WAIT
P(&_primero_que_termina);
write_vc(_termino,_i);
V(&_alguien_termino);
```

(b)

```
signal(ABORT,_handler_entrada);
P(&_activados);
P(&_sem_nomcanal);
receive_guard(&_nomcanal,&_primero_que_termina,
  &_alguien_termina,_nomvar,_i);
```

(c)

Figura 4: Constructor `ALT`: (a) Código Principal; (b) Código de un Proceso Evaluador de Guardas `WAIT`; (c) Código de un Proceso Evaluador de Guardas con Entradas desde Canales.

5 Optimizaciones

En esta sección se mencionan las optimizaciones implementadas como parte del compilador; la mayoría de ellas fueron ideadas ad hoc para el proyecto, pero algunas son estándares en la teoría de compiladores. Dichas optimizaciones se dividen básicamente entre las que se realizan en tiempo de compilación (específicamente dentro del front-end) y las que se realizan en tiempo de ejecución (consecuencia del código generado especialmente por el back-end).

Como optimizaciones implementadas en la primera etapa pueden destacarse las siguientes:

- Evaluación y propagación estática de constantes, teniendo en cuenta los aspectos de evaluación de izquierda a derecha (short circuit) y de coerción de entero a booleano impuestos por la semántica de Occam.

- Simplificación de estructuras del código fuente, eliminando estructuras de control cuyo comportamiento es predecible estáticamente (por ejemplo, un `WHILE` donde su condición evalúa estáticamente a `FALSE`) y manejando los casos de construcciones vacías. Se manejan los casos en que la eliminación de partes inútiles de código convierte a su vez en inútiles a los constructores que las encierran. Esto se hace en base a la evaluación estática de constantes.

- Simplificación de estructuras anidadas con constructores del mismo tipo, para evitar el uso excesivo de recursos atribuibles a la implementación de dichos constructores.

- Minimización del uso de variables compartidas, marcando como residentes en memoria compartida sólo a aquellas variables que probablemente se usen para comunicar resultados de un proceso hijo a su padre. Esto ocurre cuando un componente de un constructor `PAR` modifica una variable que es leída posteriormente por un proceso que se ejecuta después de que el `PAR` termina. Se descartó la alternativa de mantener todas las variables en memoria compartida porque implicaba un nivel adicional de indirección para cada acceso, más las llamadas al manejador de memoria compartida para alocarlas y desalocarlas.

- Minimización del overhead por llamadas al manejador de memoria dinámica de C durante la compilación, implementando un nivel adicional que sigue una heurística orientada a no liberar memoria hasta que sea altamente probable que no se la necesite de nuevo. Esto es utilizado internamente para hacer más eficiente el procesamiento de los bloques de declaraciones.

- Extensiones a Occam standard, con construcciones que introducen canales de comunicación con la entrada y la salida standard de UNIX, a fin de posibilitar una interacción más fluida con el usuario.

- Mensajes de error y warning destinados a facilitar el proceso de debugging de programas Occam y el uso racional de recursos, avisando de la eventual no terminación de programas, variables no utilizadas, variables no inicializadas, etc.

- Generación de código intermedio declarando identificadores sólo para aquellos que efectivamente se usan en el código Occam.

en tanto que en la segunda etapa se destacan:

- Utilización de manejadores para disminuir la cantidad de llamadas al sistema operativo aumentando la eficiencia, y también para proveer servicios de la granularidad necesaria para la ejecución del código C generado.

- Generación paralela asincrónica de procesos en tiempo de ejecución, utilizando esquema recursivo de spawning que extrae el máximo paralelismo del sistema time-sharing. Básicamente, en vez de hacer que el proceso padre genere a todos sus hijos y espere a que terminen para seguir, cada proceso genera a 0, 1 o 2 hijos y luego realiza su parte del cómputo. Debido a que la creación de un proceso involucra entrada/salida de periféricos del sistema, esta estrategia mejora la eficiencia creando los procesos según una estructura de árbol binario que aprovecha el paralelismo entre cómputo y entrada/salida.

- Minimización de la cantidad de procesos generados dinámicamente, de dos formas distintas: haciendo que el proceso padre en un constructor `PAR` ejecute una parte del código en vez de esperar a que sus hijos terminen, y evaluando sólo las guardas del `ALT` que tengan alguna posibilidad de hacerse ciertas.

- Minimización de la cantidad de procesos existentes simultáneamente en el sistema y de los tiempos de espera para sincronizar, a través de la emisión de interrupciones del proceso controlador del `ALT` a los evaluadores de guardas que los obliga a terminar tan pronto como alguna guarda se ha hecho cierta.

6 Conclusiones y Trabajo Futuro

Se desarrolló un compilador que hace dos pasadas, la primera sobre el código fuente Occam generando directamente un árbol correcto, chequeado y simplificado (sin optimizar sobre el código intermedio sino sobre el código fuente); y la segunda sobre el código intermedio generado por la primera. Se concibieron e implementaron optimizaciones que tienen impacto en la eficiencia de la ejecución del programa Occam y en el uso de recursos provistos por el sistema operativo.

Para hacer una interface al usuario más amigable, extendimos al lenguaje Occam con canales que comunican a un proceso con la entrada y la salida estándares de UNIX, en vez de comunicar a dos procesos entre sí.

Como extensión propuesta para mejorar la eficiencia del front-end, podrían implementarse al lexer y al parser como procesos concurrentes funcionando según un esquema de productor-consumidor, como un pipeline. Esto requeriría la existencia de un buffer común a ambos donde se guardaran tokens ya procesados por el lexer pero no por el parser. El overhead debido a sincronizaciones para el acceso al buffer se vería compensado por la mayor eficiencia en tiempo, ya que en un bloque físico de almacenamiento en disco caben en general muchos tokens, y así el

lexer podría cargar el siguiente bloque de disco (operación que de otra forma retardaría también al parser) mientras el parser procesara tokens extraídos del buffer común.

Referencias

[Aho et al. 86] A. Aho, R. Sethi, J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Ancilotti 87] P. Ancilotti, M. Boari. *Programmazione Concorrente*. Versione Preliminare, 1987.

[AT&T 84] AT&T. 3B2 Computer Utilities Guide Vol. II – Interprocess Communication.

[Bach 86] M. Bach. *The Design of the UNIX Operating System*. Prentice-Hall International, 1986.

[Dijkstra 65] E. W. Dijkstra. Cooperating Sequential Processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[Hoare 85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Hull 86] M. E. C. Hull. Occam - A Programming Language for Multiprocessor Systems. *Computer Languages*, Volume 12, Number 1, pp. 27-37.

[INMOS 84] INMOS Limited. *Occam Programming Manual*. Prentice-Hall International, 1984.

[Peterson 85] J. L. Peterson, A. Silberschatz. *Operating System Concepts*. Addison-Wesley, 1985.

[Stevens 90] W. R. Stevens. *UNIX Network Programming*. Prentice-Hall International, 1990.