AAP - Trabajo práctico Nº 1: Verificación de Programas

Fernando Bugni, Fernando Paulovsky, Gervasio Perez

September 23, 2011

1 Ejercicio 1

Este ejercicio nos sirvió mucho para empezar a utilizar CVC3. La resolución se encuentra en el archivo *Ejercicio1*, y la transcribimos a continuación. Utilizamos fuertemente los *FOR ALL* y la definición de variables; que esto ultimo lo vamos a utilizar seguido mas adelante.

```
% Ejercicio 1
% a
QUERY FORALL (x, y, z: INT): x <= y AND y <= z => x <= z ;

% b
QUERY FORALL (a, b: INT): a < b XOR a = b XOR a > b ;

% c
QUERY FORALL (a, b: INT): a <= b AND b <= a+1 => b = a XOR b = a + 1 ;
```

2 Ejercicios 2 y 3

En esta sección trataremos los aspectos del TP relacionados a la implementación del verificador Pest.

2.1 Consideraciones generales de diseño y algunas decisiones tomadas

En nuestro TP utilizamos extensivamente el patrón *Visitor* para las recorridas de los árboles sintácticos de programas Pest, de predicados lógicos y de términos. Nuestra implementación soporta una cantidad arbitraria de construcciones Pest anidadas.

Para la implementación de las llamadas a procedimiento decidimos asumir que las llamadas a procedimiento reciben parámetros correctos, es decir, si el procedimiento $suma(a, b)$ modifica a la variable a, invocaciones del tipo $suma(15, j)$ no están contempladas en nuestro programa (que se indefine), dado que 15 es un literal y no una variable válida modificable.

2.2 Desarrollo

2.2.1 Primer enfoque de verificación

Inicialmente optamos por implementar el verificador armando una fórmula lógica con cuantificadores, lo cual probó ser arduo especialmente con respecto al *debugging*.

Luego de la clase práctica del Miércoles 14/9 decidimos cambiar el enfoque completamente, utilizando la técnica de SSA (*Static Single Assignment*). El cambio no resultó problemático porque recién estábamos llegando a implementar el if y buena parte del código de los Visitors pudo ser reutilizada.

2.2.2 Versión final

El traductor definitivo convierte el output CVC3 en una secuencia de declaraciones de variables, y comandos ASSERT y comandos QUERY para verificar condiciones. Para ello necesita usar la técnica SSA para asegurar que cada asignación va a generar una nueva versión de un variable x siendo asignada.

Para el nuevo enfoque adoptamos el concepto de *contexto* mencionado en clase, que nos permite mantener un mapeo de nombres de variable a versiones para su uso durante la traducción.

2.3 Clases desarrolladas

Para la realización del TP desarrollamos los paquetes **budapest.pest.pesttocvc3** y **budapest.pest.predtocvc3** que contienen el código de traducción. Éstos contienen las siguientes clases:

- **PestToCVC3Translator**: Implementación de Visitor que recorre un programa Pest y lo traduce, devolviendo un String con los comandos CVC3 que lo verifican.
- **PestVarContext**: Representación de un contexto de variables Pest versionadas.
- **PredVarReplacer** y **TrmVarReplacer**: Visitors que realizan el reemplazo de variables en Predicados y en Terms respectivamente. Hacen uso de un PestVarContext de donde toman el nombre que debe usarse para cada variable en el reemplazo.
- **PredParamReplacer** y **TrmParamReplacer**: Similares a los anteriores, pero hacen un reemplazo de un nombre de variable por un Term cualquiera. Necesarios para implementar el reemplazo de variables en una llamada a procedimiento.
- **PestVarBinder**: Auxiliar para calcular el mapeo de nombres de parámetro de un procedimiento a las expresiones que se usaron para invocarlo.

2.3.1 PestToCVC3Translator

Este Visitor hace uso de contextos de variables para resolver el nombre que debe tener cada aparición de una variable al momento de usarla en una fórmula CVC. Además se ocupa de separar y unir contextos cuando el flujo del programa lo requiere:

- **A continuación de una construcción** if. Es necesario ejecutar cada rama en un contexto separado, y luego "unir" ambos contextos al contexto principal: dando un nombre nuevo a cada una de las variables que fue modificada en alguna de las dos ramas, y especificando la relación entre la condición del if y su valor final con comandos $ASSERT$.

- **A continuación de una construcción** $while$. Es necesario, por un lado, demostrar las propiedades de una iteración k del loop en un contexto separado; y luego actualizar el contexto original con las variables que fueron modificadas en el contexto del loop.

- **A continuación de una llamada a procedimiento**. Es necesario verificar el pre y asegurar el post del procedimiento llamado, instanciándolos con las parámetros pasados. Tambien es necesario actualizar el contexto. Para un ejemplo, ver la sección sobre el **PestVarBinder**.

2.3.2 PestVarContext

Esta clase es un Map extendido para guardar una referencia a un contexto padre, y además contiene una instancia de clase de un Map con la última versión usada de una variable. Esto es necesario para que contextos independientes no utilicen el mismo número de versión para nombrar a una variable.

2.3.3 PestVarBinder

Esta clase encapsula el algoritmo que calcula, en base a una llamada a procedimiento ($CallStmt$) y al $Program$ que la contiene:

- el $Procedure\ p$ objetivo de la llamada, buscándolo en el $Program$ provisto;

- el Map que traduce nombres de parámetros a las expresiones usadas como parámetros de la llamada a procedimiento;

- el conjunto de nombres de variable que fueron pasadas como parámetro y van a ser "tocadas" por el procedimiento.

Por ejemplo, para el procedimiento

```
sumaSiMenor(a,b)
:? a < b
:! a = a@pre + b@pre
:* a
{
  a <- a + b
}
```

y el fragmento de código de invocación

```
sumaSiMenor(j, k + 1)
```

se generaría un PestVarBinder con el el Map $\{a \to j,\ b \to k+1\}$, y con el conjunto de variables tocadas $\{j\}$.

Desde ya estos datos son muy útiles al **PestToCVC3Translator** y a los Visitors **PredParamReplacer** y **TrmParamReplacer** a la hora de traducir

pres y posts de procedimientos, y para actualizar el contexto que realiza una llamada a procedimiento con versiones nuevas de las variables tocadas.

Para este ejemplo, el contexto pre-llamada C_{pre} sería $[j \to j_i, k \to k_j]$, y el contexto C_{post} sería $[j- > j_{i'}, k- > k_j]$. Las pre y post se verificarían así:

```
% PRE:
QUERY ( j_i < k_j + 1 );
% POST:
j_i': INT;
ASSERT ( j_i' = j_i + k_j + 1);
```

2.4 Conclusiones acerca de la implementación

El segundo enfoque con SSA resultó muy intuitivo a la hora de escribir la verificación CVC3, y tuvimos mucho más claro cómo avanzar en la implementación. Fue interesante desarrollar las llamadas a procedimiento (aunque menos complejas que desarrollar el *while*).

En conjunto, resultó una experiencia muy interesante.