Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Traductores e Interpretadores
Trimestre Septiembre-Diciembre 2022

Etapa IV: Generación de la semántica de GCL

Esta es la última fase del proyecto y consiste en traducir un programa escrito en GCL en una fórmula que describa la semántica del programa. Para imprimir dicha fórmula es necesario usar un lenguaje para poder escribirla. En los libros del tema, la semántica (denotacional) de lenguajes de programación se escribe en el lenguaje de la teoría de conjuntos.

Puede ser de utilidad usar LaTeX para escribir fórmulas de la teoría de conjuntos, sin embargo puede ser muy complicado programar un traductor de GCL a LaTeX directamente. En su lugar los programas de GCL serán traducidos a un lenguaje intermedio, llamado *PreApp*, que es mucho más sencillo que LaTeX, para luego traducir de *PreApp* a LaTeX usando la siguiente aplicación web `http://173.230.133.51/CalcLogicTest/eval/AdminTeoremas/applicativeToLatex`.

Su entrega consistirá únicamente en implementar un traductor de GCL al lenguaje *PreApp*, pero en caso que le sea pasado un programa mal escrito, la salida esperada debe ser idéntica al de la entrega pasada. El profesor utilizará la aplicación web para traducir de *PreApp* a LaTeX, solo para que la corrección sea más sencilla de realizar, ya que será más fácil entender una fórmula renderizada con LaTeX que escrita en *PreApp*. No es necesario que usted utilice dicha aplicación web para realizar la entrega, pero le puede ser de mucha ayuda para visualizar las fórmulas en el formato de los libros, de esta forma le será más fácil validar si tiene sentido o no, lo que su traductor está traduciendo. En este mismo sentido, para hacer validaciones, puede que sea útil la siguiente aplicación web `http://173.230.133.51/CalcLogicTest/eval/AdminTeoremas/latexToApplicative` donde se puede ejecutar un traductor inverso, es decir allí usted puede traducir de LaTeX a *PreApp*.

El lenguaje *PreApp* está clasificado dentro de una categoría llamada Lenguajes Aplicativos. Una breve descripción de lo que es un lenguaje aplicativo se hace en la siguiente sección.

1. Lenguajes Aplicativos

Sean V y Σ dos alfabetos. A los símbolos de V se le llamaran símbolos variables o simplemente variables, en cambio a los símbolos de Σ se le llamaran símbolos constante o simplemente constantes. Adicionalmente a dichos alfabetos se usarán los símbolos λ, $'('$, $')'$ y el punto $'.'$. Un lenguaje aplicativo L se define como aquel lenguaje cuyas frases son subconjunto de $(V \cup \Sigma \cup \{\lambda, (,), .\})^*$ cumpliendo con la siguiente definición recursiva:

- Una variable o una constante son frases de L
- Si t_1 y t_2 son frases de L, entonces $t_1\ t_2$ es una frase de L
- Si t_1 es una frase de L, etonces (t_1) es una frase de L
- Si t_1 es una frase de L y x es una variable de V, entonces $\lambda x.t_1$ es una frase de L.

Generalmente un lenguaje aplicativo se usa para representar aplicaciones funcionales. Por ejemplo, la función f aplicada a 2, que en cálculo se escribe $f(2)$, puede ser escrita en la notación del lenguaje anterior como $f\ 2$. En el ejemplo anterior f es un símbolo variable, por lo que $f\ 2$ indica la evaluación o aplicación de 2 a una función desconocida f. Los símbolos constantes de Σ no se refieren a las constantes

como se acostumbran en cálculo, es decir no se refieren solo a un número (como en el caso de la constante 2), sino que también pueden referirse a funciones conocidas. Por ejemplo si + fuese un símbolo constante, entonces este puede ser usado para referirse a la función suma.

La idea detrás del significado de los símbolos de V y Σ, es que los símbolos de V son para referirse a objetos desconocidos, en cambio los símbolos de Σ son para referirse a símbolos conocidos, sean funciones o no. Aunque los símbolos de V se refieren a objetos desconocidos, es posible conocer el tipo de estos símbolos por el contexto. Por ejemplo si x es una variable y t_1 es cualquier término de L, entonces de la frase $x\ t_1$ se puede deducir (aunque no se sepa quién es exactamente x) que x debe ser una función porque se le está aplicando t_1. Cuando un símbolo se refiere a una función, se dice que el símbolo es de tipo $p \to q$, es decir que es una función que recibe un objeto de tipo p y devuelve un objeto de tipo q.

En los lenguajes aplicativos todas las funciones dependen de un solo argumento y no de n, sin embargo una función puede recibir una función o devolver una función. Por ejemplo, es posible que una función sea de tipo $(p \to q) \to r$ o $p \to (q \to r)$, en el primer caso recibiría una función de tipo $p \to q$ y devolvería un objeto de tipo r, y en el otro caso recibiría un objeto de tipo p y devolvería una función de tipo $q \to r$. Si los enteros se denotan con el tipo t, una forma de codificar la función suma + en un lenguaje aplicativo, es como una función de tipo $t \to (t \to t)$. Es decir, cuando la suma recibe un primer argumento $'a'$, se devuelve una función $+\ a$ que está esperando el segundo argumento para terminar la suma. De esta forma la suma de a y b se escribe en notación aplicativa como $(+\ a)\ b$. Es tradición en los lenguajes aplicativos abreviar los paréntesis asumiendo que la apliación funcional asocia a izquierda, es decir que la frase $+\ a\ b$ abrevia $(+\ a)\ b$ y nunca $+\ (a\ b)$.

El símbolo λx en un lenguaje aplicativo tiene la intención de referirse a un operador que convierte expresiones en funciones. Es decir, si x es de tipo p, y t_1 es de tipo q, entonces $\lambda x.t_1$ es una función de tipo $p \to q$. Por ejemplo, volviendo al caso de la suma, si los enteros se denotan con el tipo t y x es de tipo t, entonces la expresión $+\ x\ 2$ es de tipo t, sin embargo $\lambda x.+\ x\ 2$ es de tipo $t \to t$, es decir una función que está esperando el valor entero de la variable x para devolver la suma de x con 2.

Por último es importante hacer notar que el operador $\to$ asocia a la derecha, de esta forma se puede abreviar $p \to (q \to r)$ sin paréntesis como $p \to q \to r$.

2. Descripción del lenguaje *PreApp*

El lenguaje *PreApp* es una versión aplicativa para escribir fórmulas en la lógica de predicados. El símbolo λ se escribirá cómo `\lambda`, el alfabeto de variables es $V := \{$`x_{1}`$, $`x_{2}`$, \ldots, $`x_{n}`$, \ldots\}$ y el de las constantes es $\Sigma := \{$`c_{1}`$, $`c_{2}`$, \ldots, $`c_{n}`$, \ldots\}$. Cada constante `c_{n}` representa un operador o constante en la lógica de predicados según las tablas en la Figura 1 y 2. Dichas constantes tienen asignado un tipo, de modo tal que toda expresión booleana será de tipo b y toda expresión no booleana será de tipo t (en lógica se le llama término a toda expresión no booleana, por eso se decidió usar t para nombrar a este tipo).

Una constante `c_{i}` tiene marcado un 2 en la columna "argumentos", cuando es una función que al recibir un argumento, devuelve otra función que está esperando otro argumento, de modo que al recibir este último se devuelve una constante (no una función). Dicho de otra forma, la constante `c_{i}` tiene dos argumentos cuando `c_{i} a1 a2` es un valor constante (no una función). Note entonces, que los símbolos constantes `c_{i}` que no son funciones, son aquellos que tienen cero argumentos en la columna de argumentos. Los símbolos $a1$, $a2$ y $a3$ en una fila i de la tabla, se refieren a los argumentos 1, 2 y 3 del símbolo `c_{i}`, en caso de que éste reciba tres argumentos. En caso de que `c_{i}` reciba dos o un argumento nada más, el significado de $a1$ y $a2$ es análogo.

La columna "tipo" indica el tipo del símbolo, por ejemplo `c_{1}` representa a la equivalencia, la cual es de tipo $b \to b \to b$, es decir que recibe dos expresiones booleanas y devuelve un booleano. Análogamente, el símbolo `c_{16}`, que representa la pertenencia entre conjuntos, tiene tipo $t \to t \to b$, indicando que

id	notacion	argumentos	tipo
c_{1}	$a2 \equiv a1$	2	b->b->b
c_{2}	$a2 \Rightarrow a1$	2	b->b->b
c_{3}	$a2 \Leftarrow a1$	2	b->b->b
c_{4}	$a2 \vee a1$	2	b->b->b
c_{5}	$a2 \wedge a1$	2	b->b->b
c_{6}	$a2 \not\equiv a1$	2	b->b->b
c_{7}	$\neg a1$	1	b->b
c_{8}	$true$	0	b
c_{9}	$false$	0	b
c_{10}	$a1^{z}_{a2}$	2	(b->b)->b->b
c_{11}	$(\forall x \mid : \lambda x.a1)$	1	(t->b)->b
c_{12}	$(\forall x \mid \lambda x.a2 : \lambda x.a1)$	2	(t->b)->(t->b)->b
c_{13}	$(\exists x \mid : \lambda x.a1)$	1	(t->b)->b
c_{14}	$(\exists x \mid \lambda x.a2 : \lambda x.a1)$	2	(t->b)->(t->b)->b
c_{15}	$a2 = a1$	2	t->t->b
c_{16}	$a2 \in a1$	2	t->t->b
c_{17}	$a2 \notin a1$	2	t->t->b
c_{18}	$\emptyset$	0	t
c_{19}	$\{x \in \lambda x.a2 \mid \lambda x.a1\}$	2	(t->t)->(t->b)->t
c_{20}	$\{a1\}$	1	t->t
c_{21}	$a2, a1$	2	t->t->t
c_{22}	$\bigcup a1$	1	t->t
c_{23}	$\bigcap a1$	1	t->t
c_{24}	$a2 \cup a1$	2	t->t->t
c_{25}	$a2 \cap a1$	2	t->t->t
c_{26}	$a2 \subset a1$	2	t->t->b
c_{27}	$a2 \subseteq a1$	2	t->t->b
c_{28}	$a2 \supset a1$	2	t->t->b
c_{29}	$a2 \supseteq a1$	2	t->t->b
c_{30}	$a2 \setminus a1$	2	t->t->t

Figura 1: Tabla del significado de las constantes en *PreApp*

c_{31}	$\langle a2, a1\rangle$	2	t->t->t
c_{32}	$\langle a2 \times a1\rangle$	2	t->t->t
c_{33}	$(a1)$	1	t->t
c_{34}	$a2 \circ a1$	2	t->t->t
c_{35}	$\mathcal{P}(a1)$	1	t->t
c_{36}	$\mathbb{Z}$	0	t
c_{37}	$\mathbb{B}$	0	t
c_{38}	$a2^{a1}$	2	t->t->t
c_{39}	id_{a1}	1	t->t
c_{40}	$abort$	0	t
c_{41}	$a1^c$	1	t->t
c_{42}	0	0	t
c_{43}	1	0	t
c_{44}	2	0	t
c_{45}	3	0	t
c_{46}	4	0	t
c_{47}	5	0	t
c_{48}	6	0	t
c_{49}	7	0	t
c_{50}	8	0	t
c_{51}	9	0	t
c_{52}	$S(a1)$	1	t->t
c_{53}	$P(a1)$	1	t->t
c_{54}	$a1a2$	2	t->t->t
c_{55}	$a2 + a1$	2	t->t->t
c_{56}	$a2 - a1$	2	t->t->t
c_{57}	$a2 * a1$	2	t->t->t
c_{58}	$a3(a2 : a1)$	3	t->t->(t->t)->(t->t)
c_{59}	$[a2..a1]$	2	t->t->t
c_{60}	$-a1$	1	t->t
c_{61}	$Dom(a1)$	1	t->t
c_{62}	$a2 \neq a1$	1	t->t->t
c_{63}	$a2 < a1$	2	t->t->t
c_{64}	$a2 \leq a1$	2	t->t->t
c_{65}	$a2 > a1$	2	t->t->t
c_{66}	$a2 \geq a1$	2	t->t->t
c_{67}	π	0	t

Figura 2: Tabla del significado de las constantes en *PreApp* (segunda parte)

recibe dos expresión no booleanas y devuelve un booleano. La columna notación indica cómo luce una fórmula usando el operador `c_{i}` en el formato de los libros. Por ejemplo, la notación del símbolo `c_{1}` indica que la frase `c_{1} a1 a2` significa $a2 \equiv a1$, igualmente la notación del símbolo `c_{58}` indica que la frase `c_{58} a1 a2 a3` significa $a3(a2 : a1)$.

Si en la columna "notación" del símbolo i ocurre una subfrase como $\lambda x.a1$ o $\lambda x.a2$, entonces se está indicando que el argumento 1 o 2 debe ser una expresión lambda. Por ejemplo, como en la columna de notación para el existencial sin rango (fila 13) dice $(\exists x| : \lambda x.a1)$, entonces para representar la fórmula $(\exists x_1| : x_1 = 0)$ en *PreApp*, a sabiendas de que $x_1 = 0$ se escribe como `c_{15} c_{42} x_{1}`, se debe escribir `c_{13} (\lambda x_1 .c_{15} c_{42} x_{1})`

Por último es importante destacar que como desde el punto de vista semántico, un arreglo A es una función y la evaluación $A[i]$ es una aplicación funcional, entonces como las variables A e i se corresponden en *PreApp* a `x_{65}` y `x_{105}`, entonces $A[i]$ se escribiría como `x_{65} x_{105}`. Adicionalmente como el operador de modificación de arreglos devuelve un nuevo arreglo, entonces evaluar este nuevo arreglo en i se escribiría en notación aplicativa como $t_1\ i$, donde t_1 es el arreglo modificado. Por ejemplo, como el operador de modificación de arreglos en *PreApp* es `c_{58}` y las variables x, y, i y A se corresponden a `x_{120}`, `x_{121}`, `x_{105}` y `x_{65}` respectivamente, entonces el arreglo $A(x : y)$ se escribiría en *PreApp* como `c_{58} x_{121} x_{120} x_{65}` y la aplicación $A(x : y)[i]$ se escribiría `(c_{58} x_{121} x_{120} x_{65}) x_{105}`

3. Ejemplo

Supongamos que se tiene el siguiente programa

```
|[
   declare
      y, z : int
   y:=z+89
]|
```

Según el enunciado del proyecto la semántica de dicho programa es

$$\{x \in (\mathbb{Z} \times \mathbb{Z}) \times (\mathbb{Z} \times \mathbb{Z}) | (\exists y| : (\exists z| : x = \langle (y, z), (z + 89, z) \rangle))\} \cup \{\langle abort, abort \rangle\}$$

Sin embargo es preferible usar una versión equivalente de dicha fórmula renombrando las variables ligadas, para poder distinguir entre variables con el mismo nombre en bloques distintos:

$$\{x \in (\mathbb{Z} \times \mathbb{Z}) \times (\mathbb{Z} \times \mathbb{Z}) | (\exists x_1| : (\exists x_2| : x = \langle (x_1, x_2), (x_2 + 89, x_2) \rangle))\} \cup \{\langle abort, abort \rangle\}$$

Es decir se renombraron las variables declaradas del programa como x_i donde i es la i-ésima declaración de variables.

De esta forma su traductor debería imprimir ésta fórmula escrita en *PreApp*. Usted puede implementar esta traducción en la medida que hace un recorrido DFS sobre el AST, de forma que imprima primero la constante o variable asociada al nodo raíz, luego recursivamente imprima los hijos, y luego combine las impresiones. A continuación se explicará, sub expresión por sub expresión, como se escribe en *PreApp* la fórmula anterior.

Usando la tabla de símbolos note que:

- $\mathbb{Z} \times \mathbb{Z}$ se escribiría como `c_{32} c_{36} c_{36}`
- $(\mathbb{Z} \times \mathbb{Z}) \times (\mathbb{Z} \times \mathbb{Z})$ se escribiría como `c_{32} (c_{32} c_{36} c_{36}) (c_{32} c_{36} c_{36})`

- (x_1, x_2) se escribiría como `c_{33} ( c_{21} x_{2} x_{1})`
- $x2 + 89$ se escribiriía como `c_{33} ( c_{21} x_{2} (c_{55} (c_{54} c_{50} c_{51}) x_{2}))`
- $\langle(x_1, x_2), (x_2 + 89, x_2)\rangle$ se escribiría como
  `c_{31} (c_{33} ( c_{21} x_{2} (c_{55} (c_{54} c_{50} c_{51}) x_{2})))`
  `(c_{33} ( c_{21} x_{2} x_{1}))`
- $x = \langle(x_1, x_2), (x_2 + 89, x_2)\rangle$ se escribiría como
  `c_{15} (c_{31} (c_{33} ( c_{21} x_{2} (c_{55} (c_{54} c_{50} c_{51}) x_{2})))`
  `(c_{33} ( c_{21} x_{2} x_{1}))) x_{120}`
- $(\exists y| : z = \langle(x, y), (y + 89, y)\rangle)$ se escribiría como
  `c_{13} (\lambda x_{2} . c_{15} (c_{31} (c_{33} ( c_{21} x_{2} (c_{55}`
  `(c_{54} c_{50} c_{51}) x_{2}))) (c_{33} ( c_{21} x_{2} x_{1}))) x_{120})`
- $(\exists x| : (\exists y| : z = \langle(x, y), (y + 89, y)\rangle))$ se escribiría como
  `c_{13} (\lambda x_{1}. c_{13} (\lambda x_{2} . c_{15}`
  `(c_{31} (c_{33} ( c_{21} x_{2} (c_{55} (c_{54} c_{50} c_{51}) x_{2})))`
  `(c_{33} ( c_{21} x_{2} x_{1}))) x_{120}))`
- $\{x \in (\mathbb{Z} \times \mathbb{Z}) \times (\mathbb{Z} \times \mathbb{Z}) | (\exists x_1| : (\exists x_2| : x = \langle(x_1, x_2), (x_2 + 89, x_2)\rangle))\}$ se escribiría como
  `c_{19} (\lambda x_{120} . c_{13} (\lambda x_{1}. c_{13} (\lambda x_{2} . c_{15}`
  `(c_{31} (c_{33} ( c_{21} x_{2} (c_{55} (c_{54} c_{50} c_{51}) x_{2})))`
  `(c_{33} ( c_{21} x_{2} x_{1}))) x_{120})))`
  `(\lambda x_{120} . c_{32} (c_{32} c_{36} c_{36}) (c_{32} c_{36} c_{36}))`

4. Entrega

La entrega del proyecto es el jueves de la semana 11. Su entrega debe incluir lo siguiente:

- Un archivo comprimido `tar.gz` con el código fuente de su proyecto, debidamente documentado. El nombre del archivo deber ser **Etapa4-XX-YY.tar.gz donde XX-YY son los carné de los integrantes del grupo.**

El no cumplimento de los requerimientos podría resultar en el rechazo de su entrega.

Federico Flaviani / fflaviani@usb.ve / Diciembre 2022