R-313 Análisis de Lenguajes de Programación

Trabajo Práctico Final : SubsetL

Gianni Weinand W-0528/2

Índice

1. Lenguaje

Se presenta un DSL para manipular conjuntos finitos. El lenguaje utilizado está basado en SetL[2] y obtiene su nombre por el hecho de implementar un subconjunto de sus funcionalidades. Como ejemplo motivador, el siguiente código genera el conjunto de los cuadrados de primos hasta 1000:

```
set <int> P := {n in 2..1000 | forall d in 2..(d-1) :
                                n % d != 0 | x*x}
```

1.1. Sintaxis

SubsetL permite la declaración de constantes y funciones de un argumento. Para identificarlas se usan identificadores Id, que son los strings alfanuméricos no vacíos con al menos un caracter no numérico.

Un programa está compuesto por un único *Statement*. Al finalizar la ejecución, el intérprete muestra a qué tipos y valores ha evaluado cada constante o función.

Al declarar una constante, se debe especificar su tipo previamente. Al declarar una función, el tipo que precede a su identificador es el tipo que retorna, mientras que el tipo entre paréntesis que precede al argumento es el que le corresponde al mismo.

$Statement ::= Statement\ ;\ Statement$	*composición de statements*
$\mid Type\ Id\ :=\ Exp$	*declaración de constantes*
$\mid Type\ Id\ (Type\ Id)\ :=\ Exp$	*declaración de funciones*

Los tipos de SubsetL son:

$Type ::= \mathbf{int}$	*tipo de los enteros*
$\mid \mathbf{bool}$	*tipo de los booleanos*
$\mid set\ <Type>$	*tipo de los conjuntos*
$\mid [Type,\ Type]$	*tipo de los pares*
$\mid Type \to Type$	*tipo de las funciones*

Las expresiones de SubsetL son:

$$
\begin{array}{lr}
Exp ::= Atom & \\
\quad \mid unOp\ Exp & unOp \in UnOp \\
\quad \mid Exp\ binOp\ Exp & binOp \in BinOp \\
\quad \mid q\ IterList\ :\ Exp & q \in Quantifier
\end{array}
$$

Donde los conjuntos definidos anteriormente son:

$$
\begin{aligned}
BinOp = \{&+, -, *, /, \%, .., <, >, =, != , and, or, elem, subset, \\
&subsetEq, union, intersect, diff, cartProduct\} \\
UnOp = \{&\sim, \#, first, second\} \\
Quantifier = \{&forall, exists\}
\end{aligned}
$$

$$
\begin{array}{lr}
Atom ::= i & i \in \mathbb{Z} \\
\quad \mid b & b \in \{false,\ true\} \\
\quad \mid Id & constante \\
\quad \mid [Exp,\ Exp] & par \\
\quad \mid \{\} & conjunto\ vac\acute{i}o \\
\quad \mid \{ExpList\} & conjunto\ por\ extensión \\
\quad \mid \{IterList | Exp\} & conjunto\ por\ comprensión \\
\quad \mid \{IterList | Exp | Exp\} & conjunto\ por\ comprensión\ con\ filtro \\
\quad \mid \{ExpList\} & conjunto\ por\ extensión \\
\quad \mid Id(Id) & aplicación\ de\ función \\
\quad \mid (Exp) & paréntesis
\end{array}
$$

$$
\begin{array}{lr}
ExpList ::= Exp & singleton \\
\quad \mid Exp,\ ExpList & lista
\end{array}
$$

$$
\begin{array}{lr}
IterList ::= Id\ in\ Exp & singleton \\
\quad \mid Id\ in\ Exp,\ IterList & lista
\end{array}
$$

1.2. Semántica

Dada la gran cantidad de expresiones del lenguaje y su similaridad con las definiciones matemáticas conocidas en la teoría de conjuntos y aritmética elemental, no se dará la definición formal de cada una. En su lugar, se indicará la correspondencia con dichas definiciones. Esto permitirá un enfoque en decisiones de diseño más relevantes.

1.2.1. Scope

SubsetL tiene un solo scope global. Una vez definido, un identificador no puede ser redefinido. Las funciones y las constantes comparten el mismo espacio de identificadores. Las declaraciones de funciones no definen a su argumento, sólo reservan el nombre de la función. Es decir, el siguiente código es válido:

```
int mauro := 5;
int gradeGenerator (int mauro) := 2 * mauro;
int gradeGianniALP := gradeGenerator(mauro)
```

En cambio, los iteradores sí ligan a su identificador hasta que la mínima expresión que los contiene ha sido evaluada. Luego, el identificador vuelve a estar libre. Tampoco es posible que el mismo identificador aparezca más de una vez en una $IterList$. Se puede ver esto en los programas de ejemplo MultipleIterBindErr y RedeclarationErr. En EquivalenceRelation, se puede ver como sí es posible reutilizar un identificador en varios iteradores no anidados.

1.2.2. Reglas de Tipado

SubsetL tiene tipado estático y no tiene polimorfismo. Los *Statement* no tienen tipo. Los conjuntos contienen elementos de un solo tipo. Es por esto que en una $ExpList$, todas las expresiones deben ser del mismo tipo. En una $IterList$, todas las expresiones deben evaluar a conjuntos.

```
{IterList | Exp1 | Exp2}
```

En un conjunto por comprensión con filtro, Exp1 debe tipar a *bool*.

```
forall IterList : Exp1
exists IterList : Exp2
```

La expresión de un cuantificador (Exp1 y Exp2 en este caso) debe ser de tipo *bool*.

Todas las constantes deben evaluar al tipo con el que se las declaró. De igual manera, la expresión con la que se define una función debe evaluar al tipo con el que se la declara (asumiendo que el argumento tiene el tipo que lo acompaña). Una función sólo puede aplicarse a una expresión que evalúa al tipo de su argumento. Tristemente, las funciones sólo pueden recibir argumentos que no son funciones. Sería una extensión interesante al trabajo esta funcionalidad.

1.2.3. Semántica de Statements

```
Statement1; Statement2
```

En la composición de statements, se evalúan secuencialmente primero Statement1 y luego Statement2.

```
Type ConstId := Exp
```

En la declaración de constantes, se evalúa la expresión Exp y se verifica que su tipo sea Type. De ser así, se agrega ConstId al entrono de tipos con el valor Type y al entorno de valores con el resultado de evaluar Exp.

```
FunType FunId (ArgType ArgId) := Exp
```

En la declaración de funciones, se evalúa el tipo de Exp asumiendo que el tipo de ArgId es ArgType. Luego, se verifica que coincida con FunType. De ser así, se agrega FunId al entorno de tipos como una función que toma ArgType y retorna FunType y al entorno de valores como una función de argumento ArgId que evalúa a Exp.

1.2.4. Semántica de Expresiones

Las constantes explícitas (un número entero, el conjunto vacío, true, false) evalúan a sí mismas. Los identificadores evalúan al valor que tienen en el entorno.

```
[Exp1, Exp2]
```

Genera un par donde el primer elemento es el resultado de evaluar Exp1 y el segundo elemento es el resultado de evaluar Exp2.

```
{ExpList}
```

Genera un conjunto cuyos elementos son los resultados de evaluar cada expresión en ExpList.

```
{IterList | Exp}
```

Genera un conjunto cuyos elementos son los resultados de evaluar Exp para cada valor posible de cada iterador en IterList.

```
{IterList | FilterExp | Exp}
```

Genera un conjunto cuyos elementos son los resultados de evaluar Exp para cada valor posible de cada iterador en IterList para los cuales FilterExp evalúa a true.

```
FunId(ArgExp)
```

Busca el ArgId y la Exp correspondientes a FunId en el entorno. Evalúa Exp reemplazando previamente todas las ocurrencias de ArgId por el resultado de evaluar ArgExp. Es decir, tiene **evaluación eager**.

```
forall IterList : Exp
```

Devuelve un valor booleano. Es true si y sólo si para cada combinación de valores de los iteradores, Exp evalúa a true.

```
exists IterList : Exp
```

Devuelve un valor booleano. Es true si y sólo si existe una combinación de valores de los iteradores tal que Exp evalúa a true.

Operaciones con argumentos enteros:

- $+, -, *, /, \%$ son operadores binarios que realizan la suma, resta, producto, división entera y módulo, respectivamente.
- $<, >$ son operadores binarios que retornan un booleano indicando si los enteros satisfacen las relaciones menor y mayor respectivamente.
- $\sim$ es un operador unario que retorna el opuesto.
- .. es un operador binario que retorna el conjunto de los enteros mayores o iguales a su argumento izquierdo y menores o iguales que su argumento derecho.

Operaciones con argumentos booleanos:

- *and*, *or* son operadores binarios que realizan el and y el or lógicos, respectivamente.

Operaciones con pares como argumentos:

- $first, second$ son operadores unarios que retornan el primer y el segundo valor del par, respectivamente.

Operaciones con conjuntos como argumentos:

- # es un operador unario que retorna la cardinalidad del conjunto.
- *elem* es un operador binario que retorna un booleano indicando si su argumento izquierdo pertenece a su argumento derecho (que debe ser un conjunto del mismo tipo que el argumento izquierdo).
- $subset, subsetEq$ son operadores binarios entre conjuntos del mismo tipo que retornan un booleano indicando si satisfacen la relación subconjunto propio y subconjunto, respectivamente.
- $union, intersect, diff$ son operaciones binarias entre conjuntos de igual tipo que retornan la unión de conjuntos, la intersección de conjuntos y la diferencia de conjuntos, respectivamente.
- $cartProduct$ es un operador binario entre conjuntos que retorna el producto cartesiano entre ellos.

Operaciones con tipo variable:

- $=, != $ son operadores binarios entre argumentos de igual tipo que retorna un booleano indicando si satisfacen la relación de igualdad y diferencia, respectivamente.

2. Instalación y Manual de Uso

El intérprete de SubsetL puede encontrarse en mi repositorio de Github. Allí hay un README con las instrucciones de instalación y uso.

- Descargar el intérprete

```
$ git clone https://github.com/Weheineman/SubsetL.git
```

- Acceder a la carpeta contenedora

```
$ cd SubsetL
```

- Seguir las instrucciones del README

3. Organización del código

En la carpeta raíz se encuentran los archivos:

- `MyLexer.x` Archivo con el que Alex[4] genera el lexer. Provee los tokens.
- `MyParser.y` Archivo con el que Happy[3] genera el parser.
- `generateParser` Script de bash que usa Alex y Happy para generar el lexer y el parser y los mueve a `/src`.

En la carpeta `/src` se encuentran los módulos del intérprete. Estos son:

- `AST.hs` Define los tipos de datos utilizados así como la estructura del AST y funciones para modificarlo.
- `State.hs` Define los erroes, el entorno y la mónada de estado. Implementa funciones para modificar el entorno y lanzar errores.
- `TypeEval.hs` Implementa funciones para comprobar que un programa está bien tipado.
- `Eval.hs` Implementa funciones para evaluar un programa.
- `MyParser.hs` y `MyParser.hs` Archivos generados por Alex y Happy, respectivamente.

En la carpeta `/app` se encuentran el módulo principal:

- `Main.hs` Provee la funcionalidad de entrada y salida del intérprete.

En la carpeta `/programs` se encuentran códigos de ejemplos.

4. Detalles de Implementación

En el intérprete se usó extensamente la biblioteca Data.Set[1] para la representación interna de los conjuntos. Sus limitaciones (conjuntos homogéneos) tuvieron mucha influencia sobre el diseño de SubsetL.

5. Referencias

[1] *Data.Set documentation.*
http://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Set.html#g:1

[2] *SetL homepage.*
https://setl.org/setl/

[3] *Happy User Guide.*
https://www.haskell.org/happy/doc/html/index.html

[4] *Alex User Guide.*
https://www.haskell.org/alex/doc/html/index.html