Práctica 1: Semántica de EAB

Mario Angel García Domínguez

Facultad de Ciencias, UNAM

28 de agosto

1. Descripción del programa.

Además de implementar el lenguaje, descrito en la teoría, agregamos unas cuantas expresiones más que son: **[And], [Or], [Not], [Lt], [Gt], [Eq]**. Por lo que debemos definir su semántica dinámica y estática, para poder implementarla.

Por lo que definimos las siguientes reglas:

$$\frac{}{and(bool[b_1], bool[b_2]) \to bool[b_1 and b_2]}\ eandf \quad \frac{t_2 \to t_2'}{and(bool[b], t_2) \to and(bool[b], t_2')}\ eandd$$

$$\frac{t_1 \to t_1'}{and(t_1, t_2) \to and(t_1', t_2)}\ eandi$$

$$\frac{}{or(bool[b_1], bool[b_2]) \to bool[b_1 or b_2]}\ eorf \quad \frac{t_2 \to t_2'}{or(bool[b], t_2) \to or(bool[b], t_2')}\ eord$$

$$\frac{t_1 \to t_1'}{or(t_1, t_2) \to or(t_1', t_2)}\ eori$$

$$\frac{}{not(bool[b]) \to bool[notb]}\ enotf \quad \frac{t \to t'}{not(t) \to not(t')}\ enot$$

$$\frac{}{lt(num[n], num[m]) \to num[n < m]}\ eltf \quad \frac{t_2 \to t_2'}{lt(num[n], t_2) \to lt(num[n], t_2')}\ eltd$$

$$\frac{t_1 \to t_1'}{lt(t_1, t_2) \to lt(t_1', t_2)}\ elti$$

$$\frac{}{gt(num[n], num[m]) \to num[n > m]}\ egtf$$

$$\frac{t_2 \to t_2'}{gt(num[n], t_2) \to gt(num[n], t_2')}\ egtd$$

$$\frac{t_1 \to t_1'}{gt(t_1, t_2) \to gt(t_1', t_2)}\ egti$$

$$\frac{}{eq(num[n], num[m]) \to num[n < m]}\ eeqf$$

$$\frac{t_2 \to t_2'}{eq(num[n], t_2) \to eq(num[n], t_2')}\ eeqd$$

$$\frac{t_1 \to t_1'}{eq(t_1, t_2) \to eq(t_1', t_2)}\ eeqi$$

$$\frac{\Gamma \vdash t1 : Bool \quad \Gamma \vdash t2 : Bool}{\Gamma \vdash and(t1, t2) : Bool}\ tand$$

$$\frac{\Gamma \vdash t1 : Bool \quad \Gamma \vdash t2 : Bool}{\Gamma \vdash or(t1, t2) : Bool}\ tor$$

$$\frac{\Gamma \vdash t : Bool}{\Gamma \vdash not(t) : Bool}\ tnot$$

$$\frac{\Gamma \vdash t1 : Nat \quad \Gamma \vdash t2 : Nat}{\Gamma \vdash lt(t1, t2) : Nat}\ tlt$$

$$\frac{\Gamma \vdash t1 : Nat \quad \Gamma \vdash t2 : Nat}{\Gamma \vdash gt(t1, t2) : Nat}\ tgt$$

$$\frac{\Gamma \vdash t1 : Nat \quad \Gamma \vdash t2 : Nat}{\Gamma \vdash eq(t1, t2) : Nat}\ teq$$

Con la definición de dichas reglas la idea de la implementación queda intuitiva, pero ahí que tomar ciertas concideraciones, que en la definición no se marcan. En lo personal implementar cada función no se me hizo tan complicada la idea, pero ya al momento de escribir el código, me surgieron varios errores de diseño importante, por ejemplo, que un paso (transición) no detecta los estados bloqueados, es decir solo evalua, en un paso, por lo que los estados bloqueados los detecta como errores en tiempo de ejcución. Todos estos defectos en el diseño, no solo entorpecían el resto del programa y de la implementación, también de que alguno era indetectable por probar casos, o en algunos casos fallaban.

Otros errores, igual de diseño, si eran evidentes, pero en principio no entendía exactamente que querian las funciones en específico, o mal interpretaba su funcionamiento, por lo que mi solución tenía ciertas deficiencias, o cubrian los requerimientos según yo.

Salvo estos errores, las funciones, en general, tenían una implementación bastante intuitiva, en partícular, la función que verificaba los tipos de un programa, dado que su salida es un valor booleano, era bastante fácil de traducir de las reglas de inferencia para cada caso.

Algo que me preocupo era la cantidad de casos que analiza el Pattern Matching de *Haskell*, se me hicieron muchos casos, y un par en mi perspectiva inicial, eran redundantes. Pero me convencí de que eran necesarios dichos casos, el código queda bastante claro, en cada módulo. Son varias líneas, pero cada función va dejando en claro que hace.

En el diseño de la función **evale** en un principio no entendía como hacer para que, como evals devuelve hasta un estado bloqueado, crear esa relación entre el estado bloqueado y un error.

Término usando dos funciones auxiliares, para detectar a los estados bloqueados y a los estados finales, cada uno de manera diferente. De tal manera que de manera reducida, una función busca cualquier estado que nuestro evaluador no pueda ejecutar, y la otra si dicho estado bloqueado es ya un valor, es decir una constante booleana o un valor entero.

Entonces:

- ```
  eval1 :: Expr -> Expr
  ```

  Va a dar un solo paso según las reglas de inferencia. En caso de no poder seguir evaluando, detiene su ejecución y lanze un error. Para el caso partícilar del [**Let**], hace una evaluación perezosa y sustituye la expresión por completo.

- ```
  evals :: Expr -> Expr
  ```

  Va a evaluar en muchos pasos una expresión, hasta, llegar a un estado bloqueado. Por lo que en cada llamada es necesario verificar que sigamos teniendo transiciones. De lo contrario, devuelve el estado actual de la evaluación.

- ```
  evale :: Expr -> Expr
  ```

  Va a evaluar en muchos pasos una expresión, hasta obtener un valor, para ello, verifica que efectivamente la evaluación obtiene un valor, en el caso en que no, devuelve el error correspondiente, en tiempo de ejecución.

- Una vez que hemos definito el tipo de datos que podemos recibir, y el contexto para evaluar nuestro programa. Entonces podemos definir la función.

  ```
  vt :: TypCtxt -> Expr -> Type -> Bool
  ```

  Va a tomar un contexto, que básicamente es donde cargamos el tipo de las variables, provenientes de un [**Let**], y aplicamos las reglas que definen la semántica estática.

- ```
  eval :: Expr -> Type -> Expr
  ```

  Eval va a finalmente mezclar ambas semánticas, de manera que primero nos aseguramos que nuestra expresión va a ser del tipo que esperamos, para así poder ejecutar *evale*, pero con la garantía de no tener ningún error en tiempo de ejecución.

2. Entrada y ejecución

El programa puede ejecutarse mediante el analizador síntactico, proporcionado. O de igual manera, cargar, únicamente el módulo Semantica.hs. Y robar diferentes casos.

1. eval1:

```
*Semantic> eval1 (Add (Succ (I 5)) (I 1))
add(num[6],num[1])
```

2. evals:

```
*Semantic> evals (Let "x" (B True) (If (V "x") (I 0) (I 10)))
num[0]
```

3. evale:

```
*Semantic> evale (Add (I 5) (Mul (B True) (I 2)))
*** Exception: [Mul] expected two Integer.
*Semantic> evale (Succ (Add (I 5) (Mul (I 9) (I 9))))
num[87]
```

4. vt:

```
*Semantic> vt [] (Eq (I 0) (Add (I 5) (I 1))) Boolean
True
```

5. val:

```
*Semantic> eval (Let "y" (I 5) (Add (V "y") (I 10))) Integer
num[15]
```

3. Conclusiones.

Una práctica que puede complicarse, y la cual requiere un poco de más análisis antes de iniciar la implementación tal cual.