Lenguajes de Programación 2022-1
Nota de clase 12: TinyC
Procedimental

Javier Enríquez Mendoza

13 de enero de 2022

1 Paradigma procedimental

Hasta ahora nos hemos concentrado en el estudio y formalización de MinHs que es un lenguaje puramente funcional. Estos lenguajes son llamados puros porque su modelo de ejecución consiste únicamente en evaluar una expresión hasta hallar su valor. Sin embargo la mayoría de los lenguajes de programación de la actualidad no siguen el prototipo que hemos descrito hasta ahora.

Los modelos de ejecución de lenguajes como C o Java, consideran efectos laterales de control o almacenamiento, tales como: definición de variables globales, ejecución secuencial de instrucciones o modificaciones dinámicas a la memoria de un programa, es decir el estado de este.

El concepto de estado tiene un lugar relevante en la computación. Si creamos programas que modelen el mundo real, entonces algunos de ellos deben modelar el hecho de que en el mundo real hay eventos que lo modifican. Por ejemplo: un auto consume gasolina mientras está en marcha, de manera que un programa que modele un tanque de gasolina podría usar estados para registrar los cambios en el nivel de combustible.

Una vez que un lenguaje tiene entidades mutables o estados, es necesario hacer referencia al programa antes y después de cada mutación, es decir, el razonamiento y análisis formal de un programa depende del estado en que se ejecuta. En consecuencia se vuelve mucho más difícil determinar el comportamiento real de un programa, pues depende del momento en que se ejecuta.

El objetivo principal de esta nota es comenzar con el estudio y formalización de estos efectos laterales, para lo que se define un nuevo lenguaje con tintes procedimentales que se define como un fragmento del lenguaje de programación C, este lenguaje es llamado TinyC . Dentro de este fragmento de C se incluyen los principales constructores que nos permitirán estudiar de forma general el comportamiento de los lenguajes de programación del paradigma procedimental, como lo son: ciclos `while`, declaraciones globales y modificadores del estado.

2 Sintaxis

Comenzaremos definiendo la sintaxis para el lenguaje.

Definición 2.1 (Sintaxis Concreta de TinyC).

$$
\begin{array}{rcl}
\textit{progam} & ::= & \textit{global-decs}\ \ \textit{stmt} \\
\textit{global-decs} & ::= & \varepsilon \mid \textit{global-dec}\ \ \textit{global-decs} \\
\textit{global-dec} & ::= & \textit{fun-dec} \mid \textit{var-dec} \\
\textit{var-decs} & ::= & \varepsilon \mid \textit{var-dec}\ \ \textit{var-decs} \\
\textit{var-dec} & ::= & \textit{type}\ \textit{ident} = \textit{expr}\ ; \\
\textit{fun-dec} & ::= & \textit{type}\ \textit{ident}\ (\textit{arguments})\ \textit{stmt}\ ; \\
\textit{stmt} & ::= & \textit{expr}\ ; \mid \texttt{if}\ \textit{expr}\ \texttt{then}\ \textit{stmt}\ \texttt{else}\ \textit{stmt}\ ; \mid \texttt{if}\ \textit{expr}\ \texttt{then}\ \textit{stmt}; \\
 & & \mid \texttt{return}\ \textit{expr}\ ; \mid \{\textit{var-decs}\ \textit{stmts}\ \} \mid \texttt{while}\ (\textit{expr})\ \textit{stmt} \\
\textit{stmts} & ::= & \varepsilon \mid \textit{stmt}\ \ \textit{stmts} \\
\textit{expr} & ::= & \textit{num} \mid \textit{ident} \mid \textit{asig} \mid \textit{expr} + \textit{expr} \mid \textit{expr} - \textit{expr} \\
 & & \mid \textit{expr} > \textit{expr} \mid \textit{expr} < \textit{expr} \mid \textit{ident}\ (\textit{exprs}) \\
\textit{asig} & ::= & \textit{ident} = \textit{expr} \\
\textit{exprs} & ::= & \textit{expr} \mid \textit{expr}, \textit{exprs} \\
\textit{arguments} & ::= & \varepsilon \mid \textit{type}\ \textit{ident}, \textit{arguments} \\
\textit{type} & ::= & \mathsf{Int} \mid \mathsf{Bool}
\end{array}
$$

Para la definición de los programas del lenguaje se agrega un pequeño conjunto de constructores del lenguaje C, siguiendo las reglas de semántica que veremos en secciones siguientes se pueden formalizar fácilmente otros constructores omitidos en esta sintaxis. Para las personas conocedoras del lenguaje C, la omisión mas evidente será la de los apuntadores, esta decisión se debe a que la semántica de los apuntadores y la aritmética de apuntadores son temas complicados por si solos y quedan fuera del alcance de este curso.

En TinyC se tienen tres categorías de constructores del lenguaje:

- Declaraciones: que pueden ser tanto de variables como funciones. Se encargan de definir variables con valores asignados.
- Expresiones: que regresan un valor, estos constructores se comportan de forma similar a los constructores del paradigma funcional en el sentido de que siempre regresan un valor.
- Sentencias: son los constructores que modelan los efectos del programa en el estado, no regresan valores sino que solo se encargan de modificaciones.

Un programa es una serie de declaraciones globales seguido de una sentencia.

Observación. Es importante notar la diferencia entre las variables de TinyC con relación a las de MinHs . En MinHs las variables eran variables en un estricto sentido matemático, se pueden ligar a un valor, pero una vez que se ligan ese valor no cambia. Mientras que en TinyC las variables representan locaciones en memoria cuyo contenido puede cambiar durante la ejecución del programa. Esta diferencia sera más evidente en notas posteriores en donde se tenga un uso explicito de referencias a la memoria.

Ejemplo 2.1 (Programas en TinyC). Con la sintaxis concreta definida anteriormente se pueden construir programas como el siguiente:

```
Int result = 0;
Int div (Int a, Int b){
    Int res = 0;
    while (b < a){
        a = a - b;
        res = res + 1;
    }
    return res
};

result = div(11,5)
```

Obsérvese como de forma intuitiva este programa no regresa ningún resultado, simplemente se altera el estado en el que se ejecuta y al finalizar la ejecución la variable `result` tiene el valor 3, pero este no es el resultado de la evaluación del programa.

Observación. Para TinyC se omite la definición de la sintaxis abstracta o el proceso de análisis sintáctico por simplicidad, sin embargo es importante que quien lee esta nota entienda que expresión en sintaxis concreta corresponde a cada uno de los marcos o árboles de sintaxis abstracta que se utilizan en las siguientes secciones.

3 Semántica Operacional (Máquina $\mathcal{C}$)

Para definir la semántica dinámica de TinyC se define una nueva máquina abstracta llamada Máquina $\mathcal{C}$. Esta máquina sigue la misma estructura que las máquinas definidas para MinHs , en donde se tiene una pila de control con los cómputos pendientes y estados en donde se almacenan los valores de las variables. Para el caso de la máquina $\mathcal{C}$ no es posible el uso de sustitución como mecanismo de evaluación de variables ya que éstas dependen del estado en el que son ejecutadas.

3.1 Marcos

Comenzamos definiendo los marcos que se almacenan en la pila de control de la máquina $\mathcal{C}$. Aquí se omiten marcos para las expresiones (aquellos constructores que regresan valores) ya que pueden heredarse directamente de la máquina $\mathcal{J}$, exceptuando la llamada a función ya que su comportamiento es diferente al visto anteriormente con las aplicaciones de función.

La principal diferencia entre una llamada y una aplicación de función es que en TinyC las funciones tienen que ser declaradas para ser llamadas, mientras que en MinHs se pueden usar funciones anónimas directamente en una aplicación. Otra diferencia es que TinyC nos permite definiciones de funciones multiparamétricas mientras que en MinHs éstas se limitaban a un parámetro.

Definición 3.1 (Marcos para la pila de control de la máquina $\mathcal{C}$)**.** Se definen los siguientes marcos que usaremos en la pila de control de la máquina $\mathcal{C}$

Declaraciones

$$\frac{}{\texttt{vardec}(\mathsf{T}, x, \square)\ \ \mathsf{marco}}$$

Asignaciones

$$\frac{}{\texttt{asig}(x, \square)\ \ \mathsf{marco}}$$

Secuencia

$$\frac{}{\texttt{secu}(\square, e_2)\ \ \mathsf{marco}}$$

Condicionales

$$\frac{}{\texttt{if}(\square, e_2, e_3)\ \ \mathsf{marco}} \qquad \frac{}{\texttt{if}(\square, e_2)\ \ \mathsf{marco}}$$

Return

$$\frac{}{\texttt{return}(\square)\ \ \mathsf{marco}}$$

Llamada a función

$$\frac{}{\texttt{call}(f, \square, e_2, \ldots, e_n)\ \ \mathsf{marco}} \quad \ldots \quad \frac{}{\texttt{call}(f, v_1, v_2, \ldots, \square)\ \ \mathsf{marco}}$$

3.2 Estados

Definición 3.2 (Estados de la máquina $\mathcal{C}$)**.** Los estados de la máquina $\mathcal{C}$ son de la siguiente forma:

$$\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e \qquad\qquad \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec e$$

En donde $\mathcal{P}$ es una pila de control, $\mathcal{L}$ y $\mathcal{G}$ son ambientes de variables y e es una expresión del lenguaje.

La idea intuitiva detrás de esta definición es que cada estado tiene una pila de control con los cómputos pendientes y dos ambientes. El primero representa un ambiente auxiliar para la evaluación mientras que el segundo es el ambiente principal. El ambiente auxiliar cumple el fin de respetar el alcance de las variables dentro del cuerpo de una función, mientras que el ambiente principal se encarga de manejar el alcance de las variables declaradas de forma global. Podemos pensar en estos ambientes como un ambiente local y otro global, el momento en el que se declara una variable o una función va a definir en cuál de los ambientes se almacena.

Se definen las operaciones de consulta y modificación sobre los ambientes, necesarias para la definición de las transiciones de la máquina.

Definición 3.3 (Consulta al ambiente)**.** Definido recursivamente como sigue:

$$\frac{}{\bullet[x] = \mathsf{fail}} \qquad \frac{}{x \leftarrow v; \mathcal{E}[x] = v} \qquad \frac{}{y \leftarrow v; \mathcal{E}[x] = \mathcal{E}[x]}$$

Definición 3.4 (Modificación al ambiente)**.** Definido recursivamente como sigue:

$$\frac{}{\bullet[x \rightarrowtail v] = \mathsf{fail}} \qquad \frac{}{x \leftarrow u; \mathcal{E}[x \rightarrowtail v] = x \leftarrow v; \mathcal{E}}$$

$$\frac{}{y \leftarrow u; \mathcal{E}[x \rightarrowtail v] = y \leftarrow u; \mathcal{E}[x \rightarrowtail v]}$$

3.3 Transiciones

Ahora definimos las transiciones entre estados para la máquina $\mathcal{C}$. Estas transiciones modelan el proceso de evaluación de los programas escritos en el lenguaje.

Definición 3.5 (Transiciones de la máquina $\mathcal{C}$)**.** Las transiciones de la máquina $\mathcal{C}$ se definen en términos de los programas que se están evaluando. Como vimos en secciones anteriores en el caso de TinyC un programa no tiene un resultado final, es decir no se reduce a un valor. Por lo que se agrega un programa especifico que indica el final de la ejecución, este programa es el programa vacío y se denota como:

$$\bot$$

y define el final del proceso de evaluación de una sentencia, por lo que los estados finales de la máquina $\mathcal{C}$ son los que tienen la forma siguiente:

$$\blacklozenge \mid \mathcal{L} \mid \mathcal{G} \prec \bot$$

Con lo que se definen las transiciones con las siguientes reglas:

Declaraciones

$$\frac{}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ \mathtt{vardec}(\mathsf{T}, x, e) \longrightarrow_{\mathcal{C}} \mathtt{vardec}(\mathsf{T}, x, \square); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e}$$

$$\frac{\mathcal{G}[x] = \mathsf{fail}}{\mathtt{vardec}(\mathsf{T}, x, \square); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec v \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid x \leftarrow v; \mathcal{G} \prec \bot}$$

$$\frac{}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ \mathtt{fundec}(\mathsf{T}, f, x_1 : \mathsf{T}_1. \cdots .x_n : \mathsf{T}_n.e) \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid f \leftarrow x_1. \cdots .x_n.e; \mathcal{G} \prec \bot}$$

Asignación

$$\frac{}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ \mathtt{asig}(x, e) \longrightarrow_{\mathcal{C}} \mathtt{asig}(x, \square); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e}$$

$$\frac{\mathcal{G}[x \rightarrowtail v] = \mathcal{G}'}{\mathtt{asig}(x, \square); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec v \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid \mathcal{G}' \prec \bot}$$

$$\frac{\mathcal{G}[x \rightarrowtail v] = \mathsf{fail} \qquad \mathcal{L}[x \rightarrowtail v] = \mathcal{L}'}{\mathtt{asig}(x, \square); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec v \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L}' \mid \mathcal{G} \prec \bot}$$

Variables

$$\frac{\mathcal{G}[x] = v}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ x \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec v}$$

$$\frac{\mathcal{G}[x] = \mathsf{fail} \qquad \mathcal{L}[x] = v}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ x \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec v}$$

Secuencia

$$\frac{}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ \texttt{secu}(e_1, e_2) \longrightarrow_{\mathcal{C}} \texttt{secu}(\square, e_2); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e_1}$$

$$\frac{}{\texttt{secu}(\square, e_2); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec \bot \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e_2}$$

Condicionales

$$\frac{}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ \texttt{if}(e_1, e_2, e_3) \longrightarrow_{\mathcal{C}} \texttt{if}(\square, e_2, e_3); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e_1}$$

$$\frac{}{\texttt{if}(\square, e_2, e_3); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec \texttt{true} \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e_2}$$

$$\frac{}{\texttt{if}(\square, e_2, e_3); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec \texttt{false} \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e_3}$$

$$\frac{}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ \texttt{if}(e_1, e_2 \longrightarrow_{\mathcal{C}} \texttt{if}(\square, e_2); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e_1}$$

$$\frac{}{\texttt{if}(\square, e_2); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec \texttt{true} \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e_2}$$

$$\frac{}{\texttt{if}(\square, e_2); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec \texttt{false} \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec \bot}$$

While

$$\frac{}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ \texttt{while}(e_1, e_2) \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ \texttt{if}(e_1, \texttt{secu}(e_2, \texttt{while}(e_1, e_2)))}$$

Esta regla traduce la evaluación de un `while` a un `if` con un solo caso. Modela la siguiente regla equivalencia entre programas:

$$\texttt{while}\,(e_1)\{\,e_2\,\} \equiv \texttt{if}\,(e_1)\,\{\,e_2\,;\,\texttt{while}\,(e_1)\{\,e_2\,\}\}$$

Obsérvese como la expresión `while` sigue apareciendo en el lado derecho. De esta forma se desdobla el ciclo tantas veces como se cumpla la condición.

Llamada a función

$$\frac{}{\mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ \texttt{call}(f, e_1, \cdots, e_n) \longrightarrow_{\mathcal{C}} \texttt{call}(\square, e_1, \cdots, e_n); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ f}$$

$$\frac{}{\texttt{call}(\square, e_1, \cdots, e_n); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec v \longrightarrow_{\mathcal{C}} \texttt{call}(v, \square, e_2, \cdots, e_n); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \succ e_1}$$

$$\vdots$$

$$\frac{}{\texttt{call}(x_1 . \cdots x_n . e, v_1, \cdots, \square); \mathcal{P} \mid \mathcal{L} \mid \mathcal{G} \prec v_n \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{G} \bigstar \mathcal{L} \mid x_1 \leftarrow v_1; \cdots; x_n \leftarrow v_n; \bullet \succ e}$$

Para la evaluación de una llamada a función, primero es necesario evaluar cada uno de los parámetros con los que se llama. Una vez que todos son valores, entonces se ejecuta el cuerpo de la función usando un ambiente vacío como ambiente principal y agregando a él los parámetros de la llamada. Y guardamos el ambiente principal anterior con un símbolo especial ($\bigstar$) que sirve como separado para saber en donde termina uno y comienza el otro.

Return El constructor `return` indica el final de la ejecución de una llamada a función.

$$\frac{}{\mathcal{P} \mid \mathcal{G} \mid \mathcal{L} \succ \texttt{return}(e) \longrightarrow_{\mathcal{C}} \texttt{return}(\square); \mathcal{P} \mid \mathcal{G} \mid \mathcal{L} \succ e}$$

$$\frac{}{\texttt{return}(\square); \mathcal{P} \mid \mathcal{G} \bigstar \mathcal{L}_1 \mid \mathcal{L}_2 \prec v \longrightarrow_{\mathcal{C}} \mathcal{P} \mid \mathcal{L}_1 \mid \mathcal{G} \prec v}$$

Cuando termina la ejecución de una llamada a función. Se restaura el ambiente global y nos deshacemos del local pues solo era necesario dentro del cuerpo de la función.

Observación. El manejo de los estados hace que no todas las reglas de transición sean axiomas, lo que dificulta la implementación en comparación con las máquinas $\mathcal{H}$ y $\mathcal{J}$.

Lo importante al finalizar la ejecución de un programa en TinyC es el estado resultante, a diferencia de MinHs en donde lo importante era el valor al que se reducía.

Ejemplo 3.1 (Ejecución en la máquina $\mathcal{C}$)**.** Para mostrar el funcionamiento de la máquina $\mathcal{C}$ se evaluará el siguiente programa p de TinyC :

```
Int result = 0;
Int div (Int a, Int b){
    Int res = 0;
    while (b < a){
        a = a - b;
        res = res + 1;
    }
    return res
};

result = div(11,5)
```

$$p =_{def} \texttt{secu}(\texttt{vardec}(\mathsf{Int}, \texttt{result}, 0), \texttt{secu}(\texttt{fundec}(\mathsf{Int}, \texttt{div}, a : \mathsf{Int}, b : \mathsf{Int}, e), \texttt{asig}(\texttt{result}, \texttt{call}(\texttt{div}, 11, 5))))$$

$$e =_{def} \texttt{secu}(\texttt{vardec}(\mathsf{Int}, \texttt{res}, 0), \texttt{secu}(\texttt{while}(\dots), \texttt{return}(\texttt{res})))$$

Que se evalúa en la máquina $\mathcal{C}$ como sigue:

$$\begin{array}{rrcl}
 & \blacklozenge \mid \bullet \mid \bullet & \succ & p \\
 & & \vdots & \\
 & \blacklozenge \mid \bullet \mid \texttt{div} \leftarrow a.b.e; \texttt{result} \leftarrow 0; \bullet & \succ & \texttt{call}(\texttt{div}, 11, 5) \\
\texttt{call}(\square, 11, 5); & \blacklozenge \mid \bullet \mid \texttt{div} \leftarrow a.b.e; \texttt{result} \leftarrow 0; \bullet & \succ & \texttt{div} \\
\texttt{call}(\square, 11, 5); & \blacklozenge \mid \bullet \mid \texttt{div} \leftarrow a.b.e; \texttt{result} \leftarrow 0; \bullet & \prec & a.b.e \\
\texttt{call}(a.b.e, \square, 5); & \blacklozenge \mid \bullet \mid \texttt{div} \leftarrow a.b.e; \texttt{result} \leftarrow 0; \bullet & \succ & 11 \\
\texttt{call}(a.b.e, 11, \square); & \blacklozenge \mid \bullet \mid \underbrace{\texttt{div} \leftarrow a.b.e; \texttt{result} \leftarrow 0;}_{\mathcal{G}} \bullet & \prec & 5 \\
 & \blacklozenge \mid \mathcal{G}\bigstar \bullet \mid a \leftarrow 11; b \leftarrow 5; \bullet & \succ & e \\
 & & \vdots & \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 0; a \leftarrow 11; b \leftarrow 5; \bullet & \succ & \texttt{while}(b < a, \dots) \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 0; a \leftarrow 11; b \leftarrow 5; \bullet & \succ & \texttt{if}(a < b, \dots; \texttt{while}(b < a, \dots)) \\
 & & \vdots & \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 0; a \leftarrow 11; b \leftarrow 5; \bullet & \succ & a = a - b; res = res + 1; \texttt{while}(\dots) \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 0; a \leftarrow 6; b \leftarrow 5; \bullet & \succ & res = res + 1; \texttt{while}(\dots) \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 1; a \leftarrow 6; b \leftarrow 5; \bullet & \succ & \texttt{while}(\dots) \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 1; a \leftarrow 6; b \leftarrow 5; \bullet & \succ & \texttt{if}(a < b, \dots; \texttt{while}(b < a, \dots)) \\
 & & \vdots & \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 1; a \leftarrow 6; b \leftarrow 5; \bullet & \succ & a = a - b; res = res + 1; \texttt{while}(\dots) \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 1; a \leftarrow 1; b \leftarrow 5; \bullet & \succ & res = res + 1; \texttt{while}(\dots) \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 2; a \leftarrow 1; b \leftarrow 5; \bullet & \succ & \texttt{while}(\dots) \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 2; a \leftarrow 1; b \leftarrow 5; \bullet & \succ & \texttt{if}(a < b, \dots; \texttt{while}(b < a, \dots)) \\
\texttt{secu}(\square, \texttt{return}(\texttt{res})); \mathcal{P} & \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 2; a \leftarrow 1; b \leftarrow 5; \bullet & \prec & \bot \\
 & \mathcal{P} \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 2; a \leftarrow 1; b \leftarrow 5; \bullet & \succ & \texttt{return}(\texttt{res}) \\
 & \texttt{return}(\square); \mathcal{P} \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 2; a \leftarrow 1; b \leftarrow 5; \bullet & \succ & \texttt{res} \\
 & \texttt{return}(\square); \mathcal{P} \mid \mathcal{G}\bigstar \bullet \mid \texttt{res} \leftarrow 2; a \leftarrow 1; b \leftarrow 5; \bullet & \prec & 2 \\
 & \texttt{asig}(\texttt{result}, \square); \blacklozenge \mid \bullet \mid \texttt{div} \leftarrow a.b.e; \texttt{result} \leftarrow 0; \bullet & \prec & 2 \\
 & \blacklozenge \mid \bullet \mid \texttt{div} \leftarrow a.b.e; \texttt{result} \leftarrow 2; \bullet & \prec & \bot
\end{array}$$

En la ejecución anterior podemos observar como finaliza con el programa vacío y en el ambiente se tiene almacenado en la variable `result` el resultado que nos interesa de la ejecución del programa. Mientras que el proceso de evaluación no arrojo ningún resultado.

4 Semántica Estática

Por último se definen las reglas de semántica estática para TinyC en relación al tipado de los programas del lenguaje.

Definición 4.1 (Semántica estática de TinyC). Se define la semántica estática del lenguaje TinyC con la siguiente categoría de tipos:

$$\mathsf{T} ::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{T}_1 \to \mathsf{T}_2 \mid \mathsf{Void}$$

Sobre la cual se definen las siguientes reglas de tipado para los programas del lenguaje.

Declaraciones

$$\frac{\Gamma \vdash e : \mathsf{T}}{\Gamma \vdash \texttt{vardec}(\mathsf{T}, x, e) : \mathsf{Void}}$$

$$\frac{\Gamma, x_1 : \mathsf{T}_1, \cdots, x_n : \mathsf{T}_n, f : \mathsf{T}_1 \to \cdots \to \mathsf{T}_n \to \mathsf{T} \vdash e : \mathsf{T}}{\Gamma \vdash \texttt{fundec}(\mathsf{T}, f, x_1 : \mathsf{T}_1. \cdots .x_n : \mathsf{T}_n.e) : \mathsf{Void}}$$

Asignaciones

$$\frac{\Gamma, x : \mathsf{T} \vdash e : \mathsf{T}}{\Gamma, x : \mathsf{T} \vdash \texttt{asig}(x, e) : \mathsf{Void}}$$

Al tratarse de una asignación, es necesario que dentro del ambiente se encuentre la variable x con su tipo y se debe verificar que el nuevo valor para x tenga el mismo tipo.

Variables

$$\frac{}{\Gamma, x : \mathsf{T} \vdash x : \mathsf{T}}$$

Secuencia

$$\frac{\Gamma \vdash \texttt{vardec}(\mathsf{T}, x, e) : \mathsf{Void} \quad \Gamma, x : \mathsf{T} \vdash e_2 : \mathsf{S}}{\Gamma \vdash \texttt{secu}(\texttt{vardec}(\mathsf{T}, x, e), e_2) : \mathsf{S}}$$

$$\frac{\Gamma \vdash \texttt{fundec}(\mathsf{T}, f, x_1 : \mathsf{T}_1. \cdots .x_n : \mathsf{T}_n.e) : \mathsf{Void} \quad \Gamma, f : \mathsf{T}_1 \to \cdots \to \mathsf{T}_n \to \mathsf{T} \vdash e_2 : \mathsf{S}}{\Gamma \vdash \texttt{secu}(\texttt{fundec}(\mathsf{T}, f, x_1 : \mathsf{T}_1. \cdots .x_n : \mathsf{T}_n.e), e_2) : \mathsf{S}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Void} \quad \Gamma \vdash e_2 : \mathsf{T}}{\Gamma \vdash \texttt{secu}(e_1, e_2) : \mathsf{T}}$$

Condicionales

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \mathsf{T} \quad \Gamma \vdash e_3 : \mathsf{T}}{\Gamma \vdash \texttt{if}(e_1, e_2, e_3) : \mathsf{T}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \mathsf{Void}}{\Gamma \vdash \texttt{if}(e_1, e_2) : \mathsf{Void}}$$

Obsérvese como la expresión `if` de una sola rama no puede regresar un valor sino que sólo puede modificar el estado del programa, es decir, se trata de un comando. Esto debido a la ausencia del caso `else` que hace que no se pueda garantizar el buen comportamiento de este.

While

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \mathsf{Void}}{\Gamma \vdash \texttt{while}(e_1, e_2) : \mathsf{Void}}$$

El operador `while`es un comando, es decir, devuelve un efecto puro y se le asigna el tipo Void. Adicionalmente el cuerpo debe ser también un comando.

Llamada a función

$$\frac{\Gamma \vdash f : \mathsf{T}_1 \to \cdots \to \mathsf{T}_n \to \mathsf{T} \quad \Gamma \vdash e_1 : \mathsf{T}_1 \quad \cdots \quad \Gamma \vdash e_n : \mathsf{T}_n}{\Gamma \vdash \texttt{call}(f, e_1, \cdots, e_n) : \mathsf{T}}$$

Return

$$\frac{\Gamma \vdash e : \mathsf{T}}{\Gamma \vdash \texttt{return}(e) : \mathsf{T}}$$

Como se puede observar en las reglas anteriores, las sentencias siempre tienen tipo Void esto debido a que son constructores que no regresan valores sino que modifican el estado.

Esta definición de la semántica estática si bien correcta, es provisional ya que en notas posteriores se modificará la estructura del lenguaje, por esta razón el estudio de la seguridad del sistema de tipos se deja pendiente hasta definir la última versión de TinyC .

Referencias

[1] Keller G., O'Connor-Davis L., Class Notes from the course Concepts of programming language design, Department of Information and Computing Sciences, Utrecht University, The Netherlands, Fall 2020.

[2] Miranda Perea F., González Huesca L., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-1.

[3] Harper R., Practical Foundations for Programming Languages. Working draft, 2010.

[4] Mitchell J., Foundations for Programming Languages. MIT Press 1996.

[5] Krishnamurthi S., Programming Languages Application and Interpretation; Version 26.04.2007.