Lenguajes de Programación, 2019-I
Nota de clase 10: Paradigma Imperativo II. Semántica operacional y estática para referencias*

Favio E. Miranda Perea Lourdes del Carmen González Huesca
Facultad de Ciencias UNAM

12 de octubre de 2018

1. Memoria

Considérese una expresión de la forma $\mathsf{ref}\, v$ donde v es un valor. ¿Cómo reducimos esta expresión?, en este caso la semántica operacional necesita alojar v en una celda de memoria denotada $\mathsf{ref}\, v$, la pregunta inmediata es ¿donde se lleva a cabo tal alojamiento?. Como puede observarse la sintaxis del lenguaje no incluye expresiones para denotar al resultado de una referencia, de manera que no sólo tenemos que contestar a la pregunta de cómo evaluar una referencia sino que antes debemos extender la sintaxis para poder nombrar a los valores de tipo $\mathsf{Ref}\, \mathsf{T}$.
En conclusión necesitamos extender la sintaxis abstracta para incluir valores para referencias así como la relación de evaluación para llevar un registro de todas las referencias alojadas durante la evaluación. Dicho registro es una *memoria*.
Respecto a la implementación, para la mayoria de los lenguajes de programación la memoria es esencialmente un gran arreglo de bytes. El sistema de ejecución mantiene un registro de qué parte de este arreglo se encuentra en uso; cuando se requiere alojar una nueva celda de referencia primero se aloja en un segmento suficientemente grande de la región libre del arreglo (4 bytes para un entero, 8 para un flotante, etc), se marca este segmento para registrar que está en uso y se devuelve el índice (por lo general un entero de 32 o 64 bits) que marca el inicio del nuevo alojamiento. Estos índices son las referencias.
Para propósitos de la formalización podemos pensar que la memoria es un arreglo de valores, sin importar los diferentes tamaños de sus representaciones. Más aún, podemos suponer que los índices no son sólo números y considerar que son etiquetas tomadas de un conjunto $\mathcal{L}$ el cual es ajeno al conjunto de identificadores de variables, un elemento $\ell \in \mathcal{L}$ es una *dirección*[1] simbólica o dirección de memoria. Finalmente podemos definir una *memoria* como una función parcial $\mu : \mathcal{L} \to V$ del conjunto de direcciones $\mathcal{L}$ al conjunto de valores V.

La extensión de la sintaxis abstracta es:

*Estas notas se basan en los libros de Pierce y Krishnamurti y en material de Ralf Hinze y Eduardo Bonelli.
[1] *location*

$$\text{Expresiones} \quad e ::= \ldots \mid \ell$$

$$\text{Valores} \quad v ::= \ldots \mid \ell$$

$$\text{Memorias} \quad \mu ::= \varnothing \mid (\mu, \ell \mapsto v)$$

Observemos lo siguiente:

- Una etiqueta de dirección ℓ es una expresión y también un valor, éstas serán precisamente los valores de tipo $\mathsf{Ref\ T}$.
- Una memoria μ puede ser vacía, denotada $\varnothing$, lo cual indica que toda la memoria está libre o bien un par $(\mu, \ell \mapsto v)$ que representa a una memoria cualquiera μ a la cual se le agrega una nueva celda $\ell \mapsto v$ cuya dirección es ℓ y que aloja al valor v.
- Para asignaciones necesitamos modificar el valor de una celda de memoria, dicha operación se denota $\mu[\ell \mapsto v]$ definida como sigue:

$$\mu[\ell \mapsto v\ ](x) = \begin{cases} \mu(x) & \text{si } x \neq \ell \\ \\ v & \text{si } x = \ell \end{cases}$$

  Es decir, la memoria $\mu[\ell \mapsto v]$ coincide con la memoria μ en todas sus celdas excepto en la que tiene dirección ℓ cuyo valor es v.

1.1. Extensión de la semántica operacional

La semántica operacional se debe modificar para tomar en cuenta las memorias, puesto que ahora la evaluación de programas que utilizan alojamientos, recuperaciones o asignaciones dependen de ellas.
Para esto consideramos, ahora en lugar de expresiones aisladas, pares de expresiones y memorias $\langle \mu, e \rangle$, de manera que la relación de reducción

$$\langle \mu, e \rangle \to \langle \mu', e' \rangle$$

indica que *la expresión e usando la memoria μ se reduce a la expresión e' modificando la memoria a μ'*. Como siempre las expresiones e, e' deben ser distintas pero ahora las memorias μ, μ' podrían ser iguales lo cual significa que en ciertos pasos de evaluación la memoria no se modifica.

Además de presentar las nuevas reglas de evaluación, aquí modificamos las reglas para funciones anónimas. Se deja al lector hacer las modificaciones correspondientes a las reglas para números, booleanos y otras que se deseen agregar a la extensión.

- Funciones:

$$\frac{\langle \mu, e_1 \rangle \to \langle \mu', e_1' \rangle}{\langle \mu, e_1 e_2 \rangle \to \langle \mu', e_1' e_2 \rangle} \qquad \frac{\langle \mu, e_2 \rangle \to \langle \mu', e_2' \rangle}{\langle \mu, v_1 e_2 \rangle \to \langle \mu', v_1 e_2' \rangle}$$

$$\frac{}{\langle \mu, (\lambda x : \mathsf{T}.e) v \rangle \to \langle \mu, e[x := v] \rangle}$$

- Alojamiento:
  - Para evaluar una expresión de la forma $\mathsf{ref}\, e$ primero hay que reducir e hasta que sea un valor:
$$\frac{\langle \mu, e\rangle \to \langle \mu', e'\rangle}{\langle \mu, \mathsf{ref}\, e\rangle \to \langle \mu', \mathsf{ref}\, e'\rangle}$$
  - Una expresión de la forma $\mathsf{ref}\, v$ devuelve como valor la dirección simbólica ℓ de la celda que alojará a v, la cual se agrega a la memoria en uso:
$$\frac{\ell \notin dom(\mu)}{\langle \mu, \mathsf{ref}\, v\rangle \to \langle (\mu, \ell \mapsto v), \ell\rangle}$$
    Aquí $dom(\mu)$ denota al conjunto de direcciones $\{\ell_1, \ldots, \ell_n\}$ que figuran en la memoria μ. De manera que esta regla está agregando una celda nueva a la memoria.
- Recuperación:
  - Para evaluar una operación de recuperación $!e$ primero debe reducirse e hasta que sea un valor:
$$\frac{\langle \mu, e\rangle \to \langle \mu', e'\rangle}{\langle \mu, !\, e\rangle \to \langle \mu', !\, e'\rangle}$$
  - Una recuperación $!\ell$ se evalua al valor almacenado en la celda con dirección de memoria ℓ, el cual está dado por $\mu(\ell)$.
$$\frac{\mu(\ell) = v}{\langle \mu, !\, \ell\rangle \to \langle \mu, v\rangle}$$
- Asignación:
  - Para evaluar una asignación $e_1 := e_2$ primero es necesario reducir e_1.
$$\frac{\langle \mu, e_1\rangle \to \langle \mu', e_1'\rangle}{\langle \mu, e_1 := e_2\rangle \to \langle \mu', e_1' := e_2\rangle}$$
  - Para evaluar una asignación $\ell := e_2$ primero es necesario reducir e_2.
$$\frac{\langle \mu, e_2\rangle \to \langle \mu', e_2'\rangle}{\langle \mu, \ell := e_2\rangle \to \langle \mu', \ell := e_2'\rangle}$$
  - Una asignación $\ell := v$ causa un efecto en la memoria y devuelve un valor irrelevante $()$. El efecto consiste en eliminar el valor almacenado en la celda cuya dirección es ℓ, guardando en su lugar el valor dado v.
$$\frac{}{\langle \mu, \ell := v\rangle \mapsto \langle \mu[\ell \mapsto v], ()\rangle}$$

Veamos algunos ejemplos de evaluación con las siguientes expresiones:

$$\begin{array}{rcl} c & =_{def} & \mathsf{ref}\, 0 \\ d & =_{def} & \mathsf{ref}\, 0 \\ e_1 & =_{def} & c := suc\, 1 \\ e_2 & =_{def} & x := !\ell_1 + 3 \\ e_3 & =_{def} & \texttt{let}\ x = \mathsf{ref}(\mathsf{pred}\, 1)\ \texttt{in}\ e_1\,;e_2\ \texttt{end} \end{array}$$

- Evaluación de c en la memoria $\varnothing$.

$$\begin{array}{rcl} \langle \varnothing, c\rangle & = & \langle \varnothing, \mathsf{ref}\ 0\rangle \\ & \to & \langle (\ell_1 \mapsto 0), \ell_1\rangle \end{array}$$

- Evaluación de d en la memoria $(\ell_1 \mapsto 0)$.

$$\begin{array}{rcl} \langle (\ell_1 \mapsto 0), d\rangle & = & \langle \ell_1 \mapsto 0), d\rangle \\ & \to & \langle (\ell_1 \mapsto 0, \ell_2 \mapsto 0), \ell_2\rangle \end{array}$$

- Evaluación de e_1 en la memoria $\varnothing$.

$$\begin{array}{rcl} \langle (\varnothing, e_1\rangle & = & \langle \varnothing, c := \mathsf{suc}\ 1\rangle \\ & = & \langle \varnothing, \mathsf{ref}\ 0 := \mathsf{suc}\ 1\rangle \\ & \to & \langle (\ell_1 \mapsto 0), \ell_1 := \mathsf{suc}\ 1\rangle \\ & \to & \langle (\ell_1 \mapsto 0), \ell_1 := 2\rangle \\ & \to & \langle (\ell_1 \mapsto 2), ()\rangle \end{array}$$

- La evaluación de e_2 es imposible en cualquier memoria pues la variable x está libre.

- Evaluación de e_3 en la memoria $(\ell_1 \mapsto 3)$.

$$\begin{array}{rcl} \langle (\ell_1 \mapsto 3), e_3\rangle & \to & \langle (\ell_1 \mapsto 3), \mathtt{let}\ x = \mathsf{ref}(\mathsf{pred}\ 1)\ \mathtt{in}\ e_1\,;e_2\ \mathtt{end}\rangle \\ & \to & \langle (\ell_1 \mapsto 3), \mathtt{let}\ x = \mathsf{ref}\ 0\ \mathtt{in}\ e_1\,;e_2\ \mathtt{end}\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0), \mathtt{let}\ x = \ell_x\ \mathtt{in}\ e_1\,;e_2\ \mathtt{end}\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0),\ (e_1;e_2)[x := \ell_x]\rangle \\ & = & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0),\ c := \mathsf{suc}\ 1\,;e_2[x := \ell_x]\rangle \\ & = & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0),\ \mathsf{ref}\ 0 := \mathsf{suc}\ 1\,;e_2[x := \ell_x]\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0, \ell_2 \mapsto 0),\ \ell_2 := \mathsf{suc}\ 1\,;e_2[x := \ell_x]\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0, \ell_2 \mapsto 0),\ \ell_2 := 2\,;e_2[x := \ell_x]\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0, \ell_2 \mapsto 2),\ ()\,;e_2[x := \ell_x]\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0, \ell_2 \mapsto 2),\ e_2[x := \ell_x]\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0, \ell_2 \mapsto 2),\ \ell_x :=!\ell_1 + 3\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0, \ell_2 \mapsto 2),\ \ell_x := 3 + 3\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 0, \ell_2 \mapsto 2),\ \ell_x := 6\rangle \\ & \to & \langle (\ell_1 \mapsto 3, \ell_x \mapsto 6, \ell_2 \mapsto 2),\ ()\rangle \end{array}$$

2. Estructura de datos persistentes y efímeras

Considere la definición del siguiente tipo

$$\mathtt{PTree}\ \mathsf{T} \quad =_{def} \quad [Empty : (), Node : (left : \mathtt{PTree}\ \mathsf{T}, root : \mathsf{T}, right : \mathtt{PTree}\ \mathsf{T})]$$

Este tipo representa árboles binarios y es un tipo recursivo variante con dos opciones la primera representando al árbol vacío y la segunda representando a un árbol no vacío mediante un tipo record compuesto de tres campos a saber: el subárbol izquierdo, el elemento en la raíz y el subárbol derecho.

Si se desea crear un árbol con un sólo nodo (una hoja) podemos definir una función $leaf$ como sigue:

$$\mathsf{fun}\ \mathtt{leaf}(x : \mathsf{T}) : \mathtt{PTree}\ \mathsf{T}\ \Rightarrow\ Node\,\{left = Empty(), root = x, right = Empty()\}$$

Para insertar elementos de manera que se genere un árbol de búsqueda de números naturales podemos definir una función `insert` como sigue:

```
fun insert(x:Nat, t:PTree Nat):PTree Nat =>
 if x = t.root then t else
  case arb of
      Empty() => leaf elem
    | Node r => if x < r.root then
                   Node {left  = insert (x,r.left),
                         root  = r.root,
                         right = r.right}
                 else
                   Node {left  = r.left,
                         root  = r.root,
                         right = insert (x,r.right)}
```

De esta función podemos observar que el tipo de datos `PTree` T proporciona árboles binarios persistentes, pues al insertar un elemento a un árbol dado se hace una copia del árbol original y se modifica, quedando el árbol de entrada `t` disponible para otras operaciones. De esta manera la función `insert` vuelve a construir $n+1$ nodos del árbol donde n es la longitud del camino desde la raíz hasta la hoja recién insertada.

Mediante el uso de referencias podemos representar estructuras de datos efímeras que mejoran la eficiencia de funciones como `insert`. Veamos los detalles.

$$\begin{array}{lll} \mathtt{TreeC}\ \mathsf{T} & =_{def} & \mathsf{Ref}\,(\mathtt{ETree}\ \mathsf{T}) \\ \mathtt{ETree}\ \mathsf{T} & =_{def} & [Empty : (), Node : (left : \mathtt{TreeC}\ \mathsf{T}, root : \mathsf{T}, right : \mathtt{TreeC}\ \mathsf{T})] \end{array}$$

`ETree` T es un tipo de datos para árboles binarios efímeros. Nótese que las entradas para subárboles en realidad se refieren a celdas que guardan árboles efímeros, lo que permite definir una función de inserción de forma más eficiente. Primero definimos una función que crea hojas:

$$\mathsf{fun}\ leaf(x : \mathsf{T}) : \mathtt{ETree}\ \mathsf{T}\ \Rightarrow\ Node\,\{left = \mathsf{ref}(Empty()), root = x, right = \mathsf{ref}(Empty())\}$$

La función $leaf$ crea un árbol mediante la construcción de un record cuyo campo $root$ es el elemento dado, alojando al árbol vacío en las celdas para los subárboles izquierdo y derecho.

La función `insert` se define como:

```
fun insert(x:Nat, ct:TreeC Nat):Void =>
 if x = !ct.root then () else
  case !ct of
     Empty() => t := leaf x
```

```
| Node r => if x < r.root then
                insert (x,r.left)
              else
                insert (x,r.right)
```

En este caso la función insert toma como argumento no un árbol sino una celda que almacena a un árbol. Si el contenido de dicha celda es el árbol vacío entonces se le asigna la hoja con el elemento que se desea insertar. En el caso general la llamada recursiva inserta directamente el elemento. De manera que en la celda de entrada `ct` queda guardado el árbol con el nuevo elemento insertado. Así, los árboles de tipo `ETree` T son efímeros en el sentido de que una vez insertado un elemento el árbol viejo se pierde y es imposible de recuperar. La ventaja es que se reconstruye sólamente un nodo del árbol. Obsérve que la definición de `insert` no es la más conveniente para el caso en que `ct` guarde a un árbol vacío pues el primer condicional causará un error de ejecución ¿Cómo puede arreglarse esta situación ?

3. Semántica estática para referencias

Dado que hemos modificado la semántica dinámica considerablemente, también debemos modificar la semántica estática, por ejemplo debemos decidir cómo tipar una dirección de memoria. Cuando evaluamos una expresión que contiene direcciones concretas el tipo del resultado depende de los contenidos de la memoria inicial. Por ejemplo si evaluamos $!\,\ell_2$ en la memoria $(\ell_1 \mapsto 3, \ell_2 \mapsto ())$ el resultado es $()$ y si lo hacemos en la memoria $(\ell_1 \mapsto \mathsf{true}, \ell_2 \mapsto \lambda x : \mathsf{Void}.x)$ el resultado será $\lambda x : \mathsf{Void}.x$, en el primer caso el tipo de ℓ_2 es Void mientras que en el segundo es $\mathsf{Void} \to \mathsf{Void}$. Como el tipo cambia de acuerdo a la memoria la siguiente regla parece adecuada:

$$\frac{\Gamma \vdash \mu(\ell) : \mathsf{T}}{\Gamma \vdash \ell : \mathsf{Ref}\ \mathsf{T}}$$

Es decir si la celda denotada por ℓ guarda un valor de tipo T entonces ℓ denota a una celda de tipo $\mathsf{Ref}\ \mathsf{T}$.

Ahora bien, como el tipo inferido depende de una memoria, debemos disponer de ésta, por lo que el juicio de derivación de tipos debe incluir memorias en el contexto, por lo que debemos modificar la regla como sigue:

$$\frac{\Gamma \mid \mu \vdash \mu(\ell) : \mathsf{T}}{\Gamma \mid \mu \vdash \ell : \mathsf{Ref}\ \mathsf{T}}$$

De forma similar se modificarían las demás reglas.

Esta nueva regla presenta problemas todavía, por ejemplo la verificación de tipos es muy ineficiente, dado que calcular el tipo de una dirección ℓ requiere calcular el tipo del contenido de la celda ℓ, digamos v. Si ℓ figura muchas veces en una expresión e dicho tipo se recalculará muchas veces. Peor aún si el contenido v también contiene direcciones debemos recalcular los tipos cada vez que

aparezca una locación. Como ejemplo considérese la memoria

$$(\ell_1 \mapsto \lambda x : \mathsf{Nat}\,.999,\\ \ell_2 \mapsto \lambda x : \mathsf{Nat}\,.(!\,\ell_1)x,\\ \ell_3 \mapsto \lambda x : \mathsf{Nat}\,.(!\,\ell_2)x,\\ \ell_4 \mapsto \lambda x : \mathsf{Nat}\,.(!\,\ell_3)x,\\ \ell_5 \mapsto \lambda x : \mathsf{Nat}\,.(!\,\ell_4)x)$$

Para calcular el tipo de ℓ_5 necesitamos calcular los tipos de $\ell_1, \ldots, \ell_4$.

Peor aún, con la regla propuesta podríamos entrar en un ciclo infinito sin poder derivar el tipo deseado. Por ejemplo si tenemos la memoria

$$(\ell_1 \mapsto \lambda x : \mathsf{Nat}\,.(!\,\ell_2)x,\\ \ell_2 \mapsto \lambda x : \mathsf{Nat}\,.(!\,\ell_1)x)$$

Este tipo de ciclos surgen en la práctica, por ejemplo para construir listas ligadas haciendo que nuestro sistema de tipos pueda manejarlas. Puede observarse que los problemas surgen debido a que la regla propuesta requiere calcular repetidamente el tipo de una dirección cada vez que ésta figure en una expresión. Esto no debería ser necesario, después de todo la primera vez que se genera una dirección sabemos el tipo del valor inicial almacenado en élla. Más aún, aunque los valores almacenados pueden cambiar, siempre tendrán el mismo tipo que el valor inicial, en otro caso la operación de asignación sería inválida. En pocas palabras, el contenido de una celda puede cambiar pero siempre va a guardar valores del mismo tipo. Estos tipos pueden colectarse y mantenerse en un contexto de direcciones Σ, definidos como sigue:

$$\Sigma ::= \varnothing \mid \Sigma, \ell : \mathsf{T}$$

Al igual que en los contextos de tipos al escribir $\Sigma, \ell : \mathsf{T}$ suponemos que ℓ es una etiqueta nueva no declarada antes en Σ.
Si $\Sigma = \{\ell_1 : \mathsf{T}_1, \ldots, \ell_k : \mathsf{T}_k\}$ entonces definimos $dom(\Sigma) = \{\ell_1, \ldots, \ell_k\}$. La notación $\Sigma(\ell) = \mathsf{T}$ recupera el tipo de una dirección declarada en Σ. Es decir $\Sigma(\ell) = \mathsf{T}$ si y sólo si $\ell : \mathsf{T}$ figura en Σ.

Los nuevos juicios para el sistema de tipos necesitan un contexto de tipos Σ adicional en lugar del uso explícito de la memoria μ en uso y son de la forma

$$\Gamma \mid \Sigma \vdash e : \mathsf{T}$$

Esta relación se puede leer como *la expresión e es de tipo T bajo el contexto de variables Γ y el contexto de direcciones Σ.*

Las reglas anteriores de tipado para otras expresiones se adaptan de manera directa, como ejemplo veamos las reglas para funciones y el tipo unitario.

$$\Gamma, x : \mathsf{T} \mid \Sigma \vdash x : \mathsf{T}$$

$$\frac{\Gamma, x : \mathsf{T} \mid \Sigma \vdash e : \mathsf{S}}{\Gamma \mid \Sigma \vdash \lambda x : \mathsf{T}.e : \mathsf{T} \to \mathsf{S}}$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \mathsf{T} \to \mathsf{S} \quad \Gamma \mid \Sigma \vdash e_2 : \mathsf{T}}{\Gamma \mid \Sigma \vdash e_1\, e_2 : \mathsf{S}}$$

$$\frac{}{\Gamma \mid \Sigma \vdash () : \mathsf{Void}}$$

Las reglas que involucran referencias son:

- Si el contexto Σ le da tipo T a la dirección ℓ entonces ℓ denota a una celda de tipo $\mathsf{Ref\ T}$.

$$\frac{\Sigma(\ell) = \mathsf{T}}{\Gamma \mid \Sigma \vdash \ell : \mathsf{Ref\ T}}$$

  o equivalentemente $\Gamma \mid \Sigma,\ \ell : \mathsf{T} \ \vdash \ell : \mathsf{Ref\ T}$

- Si creamos una referencia a una expresión de tipo T entonces dicha referencia tiene tipo $\mathsf{Ref\ T}$.

$$\frac{\Gamma \mid \Sigma \vdash e : \mathsf{T}}{\Gamma \mid \Sigma \vdash \mathsf{ref}\ e : \mathsf{Ref\ T}}$$

- Si e tiene tipo $\mathsf{Ref\ T}$ entonces al recuperar el valor de la referencia e éste debe tener tipo T

$$\frac{\Gamma \mid \Sigma \vdash e : \mathsf{Ref\ T}}{\Gamma \mid \Sigma \vdash\, !\, e : \mathsf{T}}$$

- Si e_1 denota una celda de tipo $\mathsf{Ref\ T}$ entonces podemos almacenar e_2 en dicha celda siempre y cuando e_2 sea también de tipo T.

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \mathsf{Ref\ T} \quad \Gamma \mid \Sigma \vdash e_2 : \mathsf{T}}{\Gamma \mid \Sigma \vdash e_1 := e_2 : \mathsf{Void}}$$

4. Seguridad del sistema

Debemos reformular nuestras propiedades de preservación y progreso en el marco de los tipos referencia. Si empezamos con la preservación un primer intento sería

$$\text{Si } \Gamma \mid \Sigma \vdash e : \mathsf{T} \text{ y } \langle \mu, e \rangle \to \langle \mu', e' \rangle \text{ entonces } \Gamma \mid \Sigma \vdash e' : \mathsf{T}$$

Esta versión es ingenua, no existe relación entre Σ y μ de manera que no podemos esperar que la semántica operacional respete los tipos de las direcciones. Por ejemplo se cumplen

- $\varnothing \mid \ell : \mathsf{Nat} \vdash\, !\, \ell : \mathsf{Nat}$
- $\langle (\ell \mapsto \mathsf{true}), !\, \ell \rangle \to \langle (\ell \mapsto \mathsf{true}), \mathsf{true} \rangle$

y claramente $\Gamma \mid \ell : \mathsf{Nat} \vdash \mathsf{true} : \mathsf{Nat}$ es inválido.

De manera que para enunciar la seguridad del sistema primero tenemos que garantizar que los contextos de direcciones Σ son *compatibles* con las memorias μ.

Definición 1 *Una memoria μ es compatible con el contexto de direcciones Σ si*

- *$dom(\mu) = dom(\Sigma)$. Es decir, si ambos la memoria y el contexto de direcciones incluyen a las mismas etiquetas y además*
- *$\Gamma \mid \Sigma \vdash \mu(\ell) : \Sigma(\ell)$ para cualquier dirección $\ell \in dom(\mu)$. Es decir, el tipo del valor $\mu(\ell)$ almacenado en la celda con dirección ℓ, debe ser el mismo que el contexto Σ predice.*

En tal caso escribimos $\Sigma \sim \mu$

Con este concepto podemos reformular la preservación como sigue:

$$\text{Si } \Gamma \mid \Sigma \vdash e : \mathsf{T},\ \Sigma \sim \mu \text{ y } \langle \mu, e\rangle \to \langle \mu', e'\rangle \text{ entonces } \Gamma \mid \Sigma \vdash e' : \mathsf{T}$$

Esto es casi correcto pero no contempla la posibilidad de que el contexto de tipos Σ haya crecido debido a que la semántica operacional puede hacer crecer la memoria, por ejemplo al crear nuevas referencias. Teniendo en cuenta esta observación finalmente podemos enunciar la preservación de tipos.

Proposición 1 (Preservación) *Si $\Gamma \mid \Sigma \vdash e : \mathsf{T},\ \Sigma \sim \mu$ y $\langle \mu, e\rangle \to \langle \mu', e'\rangle$ entonces existe $\Sigma' \supseteq \Sigma$ tal que $\Gamma \mid \Sigma' \vdash e' : \mathsf{T}$ y $\Sigma' \sim \mu'$.*

Con respecto al progreso, también debemos generalizar para tomar en cuenta memorias y contextos de direcciones.

Proposición 2 (Progreso) *Si e es una expresión tal que $\varnothing \mid \Sigma \vdash e : \mathsf{T}$, es decir, e no tiene variables libres. Entonces e es un valor o bien para cualquier memoria μ tal que $\Sigma \sim \mu$ existen una expresión e' y una memoria μ' tales que $\langle \mu, e\rangle \to \langle \mu', e'\rangle$.*

5. El ciclo while

Para terminar con la definición de un lenguaje núcleo para el paradigma imperativo necesitamos de un operador de iteración, en este caso el ciclo while.

- Sintaxis concreta: `while` e_1 *do* e_2 `end_while`
- Sintaxis abstracta: $\mathsf{while}(e_1, e_2)$
- Semántica estática: el ciclo `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:

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \mid \Sigma \vdash e_2 : \mathsf{Void}}{\Gamma \mid \Sigma \vdash \mathsf{while}(e_1, e_2) : \mathsf{Void}}$$

- Semántica dinámica: obsérvese que sólo existe una regla, la evaluación del while simplemente deja la memoria igual y evalua un condicional que nuevamente incluye al while en el caso en que la guardia sea verdadera.

$$\frac{}{\langle \mu, \mathsf{while}(e_1, e_2)\rangle \to \langle \mu, \mathsf{if}\ e_1\ \mathsf{then}\ e_2; \mathsf{while}(e_1, e_2)\ \mathsf{else}\ ()\rangle}$$

  Con esta definición es claro que se cumple lo siguiente:

  - Si $\langle \mu, e_1\rangle \to^\star \langle \mu', \mathsf{true}\rangle$ y $\langle \mu', e_2\rangle \to^\star \langle \mu'', ()\rangle$ entonces

    $$\langle \mu, \mathsf{while}(e_1, e_2)\rangle \to^\star \langle \mu'', \mathsf{while}(e_1, e_2)\rangle$$

  - Si $\langle \mu, e_1\rangle \to^\star \langle \mu', \mathsf{false}\rangle$ entonces $\langle \mu, \mathsf{while}(e_1, e_2)\rangle \to^\star \langle \mu', ()\rangle$

Se observa que la semántica operacional del while es como un proceso de eliminación de azucar sintáctica pero no del todo puesto que el while no desaparece. Esto lleva a pensar en definir al ciclo while como una función recursiva.

```
fun while(e1:Bool, e2:Void):Void => if e1 then
                                       e2; while(e1,e2)
                                    else
                                       void
```

Una vez que se tiene un ciclo primitivo como while tenemos un nucleo del paradigma imperativo y de hecho tenemos un lenguaje Turing-completo[2]. Terminemos esta sección con algunos programas que discutiremos en clase:

- Ciclo infinito:

  ```
  while true do ()
  ```

- Factorial:

  ```
  fun fact(n:Nat):Nat => let x = ref n
                             y = ref 1
                         in
                          while !x > 0 do
                           y := !x * !y;
                           x := !x - 1
                          end_while ;
                          !y
                         end
  ```

- Aproximación del logaritmo de z:

[2]Es posible probar que la presencia del ciclo while, del enunciado de asignación := y del operador de secuencia ; en un lenguaje de programación resulta en un lenguaje con poder equivalente al de las máquinas de Turing

```
x := 1;
y := 0;
while !x < z do
 x := !x + !x;
 y := !y + 1
end_while
```

- Predecesor:

```
p := 0;
m := 0;
while !m <> n do
 p := !m;
 m := !m + 1
end_while
```

- División:

```
q := 0;
r := m;
while r >= n do
 q := !q + 1;
 r := !r - n;
end_while
```

- Máximo común divisor de m, n:

```
while n <> 0 do
 r := m;
  while r >= n do
    r := !r - n;
  end_while;
 m := n;
 n := !r;
end_while
```

6. Ejercicios

1. Se desea extender el lenguaje con referencias con las siguientes operaciones:

   - eqr que decide si dos referencias de memoria son iguales.
   - eqrv que decide si dos referencias de memoria contienen el mismo valor.

   Defina las semánticas considerando a estas operaciones como primitivas. ¿ Es posible definirlas como azucar sintáctica?

2. Defina el ciclo repeat[3] como primitivo y como azucar sintáctica utilizando el ciclo while.

3. Defina el ciclo for como primitivo y como azucar sintáctica.

4. Defina una extensión con tipos productos mutables con las siguientes operaciones

   a) mkpair crea un par mutable.

   b) mfst primera proyección de un par mutable.

   c) msnd segunda proyección de un par mutable.

   d) setfst asignación a la primera componente de un par.

   e) setsnd asignación a la segunda componente de un par.

5. Defina la extensión del ejercicio anterior como azucar sintáctica usando tipos producto y referencias.

6. Defina las operaciones aritméticas usuales de manera imperativa (la exponenciación está en la nota 1 de clase). Demuestre usando la semántica formal algunos casos sencillos de evaluación.

7. Muestre la ejecución formal de algunos casos sencillos de los programas ejemplo de esta nota.

[3]Conocido también como do-while