# El lenguaje Iswim

Daniel Fridlender, Héctor Gramaglia*

June 6, 2024

Este lenguaje fue introducido por Peter Landin, y aunque nunca fue implementado, fue el primero en su tipo en ser presentado con una clara comprensión de los principios que subyacen en su diseño.

El lenguaje Iswim propone incorporar una componente imperativa a un lenguaje aplicativo eager mediante la incorporación de las referencias como un tipo más de valores, y que por lo tanto pueden ser devueltos por una función, o pasados como valor que recibe una función. Este principio es incorporado en los lenguajes Algol 68, Basel, Gendaken y Standard ML.

El lenguaje Iswim extiende al lenguaje aplicativo eager mediante las siguientes construcciones:[1]

$$\langle exp \rangle ::= \textbf{ref}\ \langle exp \rangle \mid \textbf{val}\ \langle exp \rangle \mid \langle exp \rangle := \langle exp \rangle \mid \langle exp \rangle =_{ref} \langle exp \rangle$$

La expresión **ref** $e$ extiende el estado generando una locación nueva que aloja el valor producido por $e$, Note que no hay restricciones (al menos en la sintaxis) para el valor que puede adquirir $e$. La expresión **val** $e$ estará definida cuando $e$ produzca un valor de tipo referencia, y la expresión completa devolverá lo alojado en esa referencia. La expresión $e := e'$ produce la modificación en el estado producto de la asignación ($e$ debe producir un valor de tipo referencia), devolviendo un valor especial que llamaremos **unit**; elegir devolver ese valor es arbitrario, sería razonable devolver la referencia asignada o el valor que se le asigna.

Note que típicas construcciones del lenguaje imperativo como la iteración y la declaración de variables locales no se incorporan a la sintaxis abstracta. En efecto, el orden de evaluación eager permite obteneralas como azucar sintáctico. Definiremos estas expresiones después de dar los significados denotacionales y operacionales del lenguaje.

## Semántica Denotacional

Utilizaremos para Iswim el mismo modelo de memoria utilizado para el lenguaje imperativo con procedimientos y expresiones enriquecidas. El conjunto de val-

*Iván Renison señaló algunos errores.

[1] En alguna versión anterior del apunte se agregaba explícitamente **skip**.

ores se extiende de la siguiente manera:

$$\begin{aligned} V &= V_{int} + V_{bool} + V_{fun} + V_{ref} + V_{tuple} \\ D &= (\{\mathbf{error}, \mathbf{typeerror}\} + \Sigma \times V)_\perp \\ Env &= \langle var \rangle \to V \end{aligned}$$

Respecto al predominio $V$ usado para el lenguaje aplicativo eager puro, además de agregar el predominio $V_{ref}$ varía el dominio $V_{fun}$ usado para la denotación de funciones: ahora toma el valor correspondiente al argumento y también el estado eventualmente modificado por la semántica del argumento.

$$\begin{aligned} V_{int} &= \mathbb{Z} & \iota_{int} &\in V_{int} \to V \\ V_{bool} &= \mathbb{B} & \iota_{bool} &\in V_{bool} \to V \\ V_{ref} &= Rf & \iota_{ref} &\in V_{ref} \to V \\ V_{fun} &= \Sigma \times V \to D & \iota_{fun} &\in V_{fun} \to V \\ V_{typle} &= V^* & \iota_{tuple} &\in V^* \to V \end{aligned}$$

La función $\iota_{norm}$ aceptará ahora un par estado-valor: $\iota_{norm}\langle \sigma, z \rangle \in D$. Además de valores con un estado tenemos los elementos $err, tyerr \in D$ que constituyen la denotación de los errores. Además definimos convenientemente $\langle\rangle_V = \iota_{tuple}\langle\rangle$.

Las funciones que asisten la transferencia de control se adaptan a la nueva funcionalidad: Si $f \in \Sigma \times V \to D$, entonces $f_* \in \Sigma \times D \to D$ se define:[2]

$$\begin{aligned} f_*(\iota_{norm}\,\langle \sigma, z \rangle) &= f(\langle \sigma, z \rangle) \\ f_*(err) &= err \\ f_*(tyerr) &= tyerr \\ f_*(\perp) &= \perp \end{aligned}$$

Por otro lado, si $f \in \Sigma \times V_{int} \to D$, entonces $f_{int} \in \Sigma \times V \to D$ se define:

$$\begin{aligned} f_{int}(\langle \sigma, \iota_{int}\, k \rangle) &= f(\langle \sigma, k \rangle) \\ f_{int}(\langle \sigma, \iota_\theta\, z \rangle) &= tyerr \qquad \text{si } \theta \neq int \end{aligned}$$

De manera similar se definen los operadores $(\_)_{bool}$, $(\_)_{fun}$, etc.

La función semántica será de tipo:

$$[\![\_]\!] \in \langle exp \rangle \to Env \to \Sigma \to D$$

[2]Esto corresponde a la acción en funciones del funtor $F\colon$ **PreDom** $\to$ **Dom** dado por $F(X) = (\{\mathbf{error}, \mathbf{typeerror}\} + \Sigma \times X)_\perp$.

La semántica de los construcciones típicamente imperativas es la siguiente:

$$[\![\mathbf{val}\ e]\!]\eta\sigma = \Big(\lambda\langle\sigma', r\rangle.\ \begin{cases}\iota_{norm}\,\langle\sigma', \sigma' r\rangle & \text{si } r \in dom(\sigma') \\ err & \text{si no}\end{cases}\Big)_{ref*}([\![e]\!]\eta\sigma)$$

$$[\![\mathbf{ref}\ e]\!]\eta\sigma = (\lambda\langle\sigma', z\rangle.\ \iota_{norm}\,\langle[\sigma' \mid r_{\sigma'} : z], \iota_{ref}\, r_{\sigma'}\rangle)_*([\![e]\!]\eta\sigma)$$
$$\text{donder}_{\sigma'} = new(\sigma')$$

$$[\![e := e']\!]\eta\sigma = (\lambda\langle\sigma', r\rangle.\ (\lambda\langle\hat{\sigma}, z\rangle.\ \iota_{norm}\,\langle[\hat{\sigma} \mid r : z], \langle\rangle_V\rangle)_*[\![e']\!]\eta\sigma'))_{ref*}([\![e]\!]\eta\sigma)$$

$$[\![e =_{ref}\ e']\!]\eta\sigma = (\lambda\langle\sigma', r\rangle.\ (\lambda\langle\hat{\sigma}, r'\rangle.\ \iota_{norm}\,\langle\hat{\sigma}, \iota_{bool}\, r = r'\rangle)_{ref*}$$
$$([\![e']\!]\eta\sigma'))_{ref*}([\![e]\!]\eta\sigma)$$

La semántica de las construcciones aplicativas requiere solamente adaptar la funcionalidad. La semántica de 0 o **true**, es trivial, salvo que hay que promover el resultado para que sea un elemento de $D$, y el estado por supuesto no se modifica:

$$[\![0]\!]\eta\sigma = \iota_{norm}\,\langle\sigma, \iota_{int}\, 0\rangle$$
$$[\![\mathbf{true}]\!]\eta\sigma = \iota_{norm}\,\langle\sigma, \iota_{bool}\, T\rangle$$

Para evaluar $-e$ se evalúa $e$ y se chequea que dé entero (en caso contrario, el subíndice int se encargará de disparar un error de tipos) y que no se haya producido ya algún error (en cuyo caso, el subíndice * se encargará de propagarlo). Si todo anda bien se devuelve el entero correspondiente promoviendolo para que sea un $D$.

$$[\![-e]\!]\eta\sigma = (\lambda\langle\sigma', i\rangle.\ \iota_{norm}\,\langle\sigma', \iota_{int}\ -i\rangle)_{int*}([\![e]\!]\eta\sigma)$$

Un recurso similar sirve para los operadores binarios:

$$[\![e + e']\!]\eta\sigma = (\lambda\langle\sigma', i\rangle$$
$$(\lambda\langle\hat{\sigma}, j\rangle.\ \iota_{norm}\,\langle\hat{\sigma}, \iota_{int}\, i + j\rangle)_{int*}([\![e']\!]\eta\sigma'))_{int*}([\![e]\!]\eta\sigma)$$

Los operadores típicos del cálculo lambda se adaptan como sigue:

$$[\![v]\!]\eta\sigma = \iota_{norm}\,\langle\sigma, \eta\ v\rangle$$
$$[\![e\, e']\!]\eta\sigma = (\lambda\langle\sigma', f\rangle.\ f_*([\![e']\!]\eta\sigma'))_{fun*}([\![e]\!]\eta\sigma)$$
$$[\![\lambda v.\ e]\!]\eta\sigma = \iota_{norm}\,\langle\sigma, \iota_{fun}\,(\lambda\langle\sigma', z\rangle.\ [\![e]\!][\eta \mid v : z]\sigma')\rangle$$

Finalmente, la semántica del operador de recursión se define usando el operador de menor punto fijo en el dominio $V_{fun}$:

$$[\![\mathbf{letrec}\ w = \lambda v.\, e\ \mathbf{in}\ e']\!]\eta\sigma = [\![e']\!][\eta \mid w : \iota_{fun}\, g]\sigma$$

donde

$$g\colon V_{fun} \qquad\qquad F\colon V_{fun} \to V_{fun}$$
$$g = \mathbf{Y}_{V_{fun}} F \qquad\qquad F(f)(\langle\sigma', z\rangle) = [\![e]\!][\eta | w : \iota_{fun}\, f | v : z]\sigma'$$

## Semántica operacional

Para dar la semántica denotacional sumamos como formas canónicas a las referencias:

$$\langle cnf \rangle ::= \ldots \mid Rf$$

La relación de evaluación $\_ \Rightarrow \_ \subseteq (\Sigma \times \langle exp \rangle) \times (\langle cnf \rangle \times \Sigma)$ tendrá la siguiente forma, que refleja no sólo el cómputo de un valor sino además la modificación del estado y donde además tomamos como punto de partida una expresión junto con un estado:

$$\sigma, e \Rightarrow z, \sigma'$$

Todas las reglas aplicativas del lenguaje eager se incorporan con la indicación explícita de cómo se transforma el estado. Por ejemplo la regla de la aplicación

$$\frac{e \ \Rightarrow_E \ \lambda v.e_0 \qquad e' \ \Rightarrow_E \ z' \qquad (e_0/v \mapsto z') \ \Rightarrow_E \ z}{e e' \ \Rightarrow_E \ z}$$

se transforma en:

$$\frac{\sigma, e \ \Rightarrow_E \ \lambda v.e_0, \sigma' \qquad \sigma', e' \ \Rightarrow_E \ z', \sigma'' \qquad \sigma'', (e_0/v \mapsto z') \ \Rightarrow_E \ z, \hat{\sigma}}{\sigma, e\, e' \ \Rightarrow_E \ z, \hat{\sigma}}$$

De manera similar se transforman las demás reglas del lenguaje aplicativo eager. Las nuevas reglas que describen el comportamiento de las construcciones típicamente imperativas son las siguientes:

$$\frac{\sigma, e \Rightarrow r, \sigma' \qquad \sigma', e' \Rightarrow z', \hat{\sigma}}{\sigma, e := e' \Rightarrow z', [\hat{\sigma} \mid r : z']}$$

$$\frac{\sigma, e \Rightarrow z, \sigma'}{\sigma, \mathbf{ref}\ e \Rightarrow r, [\sigma' \mid r : z]}\ r = new(\sigma')$$

$$\frac{\sigma, e \Rightarrow r, \sigma'}{\sigma, \mathbf{val}\ e \Rightarrow \sigma'\, r, \sigma'}\ r \in dom(\sigma')$$

$$\frac{\sigma, e \Rightarrow r, \sigma' \qquad \sigma', e' \Rightarrow r', \hat{\sigma}}{\sigma, e =_{ref} e' \Rightarrow \lfloor r = r' \rfloor, \hat{\sigma}}$$

## Algunas propiedades del fragmento imperativo

Dado que una frase de tipo $\langle exp \rangle$ tiene el potencial de simultaneamente producir un efecto en el estado y a su vez un valor, podemos modelar la secuencia de comandos a través de un let en el cual el valor producido por la expresión se descarte:

$$e; e' \ =_{def} \ \mathbf{let}\ v = e\ \mathbf{in}\ e' \qquad\qquad (v \notin FV\ e')$$

La semántica operacional nos permite verificar el significado esperado:

$$\frac{e, \sigma \Rightarrow z, \sigma' \qquad e', \sigma' \Rightarrow \hat{z}, \hat{\sigma}}{e; e', \sigma \Rightarrow \hat{z}, \hat{\sigma}}$$

Note que la regla pone en evidencia que el valor $z$ producido al evaluar $e$ es descartado.

Por supuesto que esta regla debe ser deducida de las reglas dadas, en tanto $e; e$ es introducido como una abreviatura. Expresándonos con precisión, es necesario probar una propiedad que exprese la regla de la siguiente manera:

**Propiedad 1.** *Si* $\sigma, e \Rightarrow z, \sigma'$ *y* $\sigma', e' \Rightarrow z', \hat{\sigma}$, *entonces* $\sigma, e; e' \Rightarrow z', \hat{\sigma}$.

Es un buen ejercicio verificar esta propiedad usando la regla de la aplicación, y el hecho de que **let** $v = e$ **in** $e'$ es una abreviatura de $(\lambda v.\ e')\, e$.

También podemos deducir inmediatamente una ecuación semántica para el punto y coma:

$$[\![e; e']\!]\eta\sigma = (\lambda\langle\sigma', z\rangle.\ [\![e']\!]\eta\sigma')_*([\![e]\!]\eta\sigma)$$

Nuevamente la ecuación refleja el hecho de que el valor $z$ producido por la evaluación de $e$ es descartado. En efecto, estamos usando la propiedad $[\![e']\!]\eta = [\![e']\!][\eta|v : z]$, garantizada por el teorema de coincidencia y por la condición $v \notin FV\ e'$. Note que aquí es fundamental el orden de evaluación eager para que la secuencia de ejecución de los comandos involucrados se respete.

En el lenguaje imperativo la declaración de variables, con **newvar**, la podíamos pensar como una modificación local a la variable declarada. En Iswim podemos simular lo mismo pero agregando una referencia que será inalcanzable desde fuera del alcance del let.

$$\mathbf{newvar}\, v := e\, \mathbf{in}\, e' \ =_{def}\ \mathbf{let}\ v = \mathbf{ref}\ e\ \mathbf{in}\ e'$$

La regla resultante de esta definición es la siguiente (que debe ser probada):

$$\frac{\sigma, e \Rightarrow z, \sigma' \qquad [\sigma'|r : z], (e'/v \mapsto r) \Rightarrow z', \hat{\sigma}}{\sigma, \mathbf{newvar}\, v := e\, \mathbf{in}\, e' \Rightarrow z', \hat{\sigma}}\ (r = new(dom(\sigma')))$$

El significado denotacional de la definición de newvar se revela en la siguiente propiedad.

**Propiedad 2.** *Si* $[\![e]\!]\eta\sigma = \iota_{norm}\langle\sigma', z\rangle$ *y* $r = new(dom\ \sigma')$ *entonces*

$$[\![\mathbf{newvar}\, v := e\, \mathbf{in}\, e']\!]\eta\sigma = [\![e']\!][\eta|v : \iota_{ref} r][\sigma'|r : z]$$

Si definimos **skip** como la tupla vacía podemos definir iteración como un tipo especial de declaración de función recursiva:

$$\mathbf{while}\ e\ \mathbf{do}\ e' \ =_{def}\ \mathbf{letrec}\ w = \lambda v.\ \mathbf{if}\ e\ \mathbf{then}\ e'; w\,\langle\rangle\ \mathbf{else\ skip\ in}\ w\,\langle\rangle$$

Aquí las variables $w$ y $v$ no deben ocurrir en $e$ ni $e'$. El verdadero sentido de la misma está dado por las reglas resultantes:

$$\frac{e, \sigma \Rightarrow \mathbf{false}, \sigma'}{\mathbf{while}\ e\ \mathbf{do}\ e', \sigma \Rightarrow \langle\rangle, \sigma'}$$

$$\frac{e, \sigma \Rightarrow \mathbf{true}, \sigma' \qquad e'; \mathbf{while}\ e\ \mathbf{do}\ e', \sigma' \Rightarrow z', \hat{\sigma}}{\mathbf{while}\ e\ \mathbf{do}\ e', s \Rightarrow z', \hat{\sigma}}$$

Para terminar la sección vamos a demostrar formalmente la validez de estas últimas. Sea $e_{while} = \mathbf{if}\ e\ \mathbf{then}\ e'; w\,\langle\rangle\ \mathbf{else\ skip}$. Entonces

$$\mathbf{while}\ e\ \mathbf{do}\ e' = \mathbf{letrec}\ w = \lambda v.\ e_{while}\ \mathbf{in}\ w\,\langle\rangle$$

Como es usual, denotemos

$$e^*_{while} = \mathbf{letrec}\ w = \lambda v.\ e_{while}\ \mathbf{in}\ e_{while}$$

Entonces, evaluar la expresión original es equivalente a evaluar:

$$(w\,\langle\rangle)/w \mapsto \lambda v.\ e^*_{while} \ = \ (\lambda v.\ e^*_{while})\,\langle\rangle$$

Luego, evaluar $\mathbf{while}\ e\ \mathbf{do}\ e'$ es lo mismo que evaluar

$$(\lambda v.\ e^*_{while})\mathbf{skip} \qquad (1).$$

Planteamos entonces la regla de la aplicación:

$$\frac{\sigma, (\lambda v.\ e^*_{while}) \Rightarrow (\lambda v.\ e^*_{while}), \sigma \qquad \sigma, \mathbf{skip} \Rightarrow \langle\rangle, \sigma \qquad \sigma, (e^*_{while}/v \to \langle\rangle) \Rightarrow ?, ?}{\sigma, (\lambda v.\ e^*_{while})\mathbf{skip} \Rightarrow ?, ?}$$

Pero note que

$$e^*_{while}/v \to \langle\rangle \ = \ e^*_{while}$$

y apara evaluarlo debemos evaluar

$$\begin{aligned} e_{while}/w \mapsto \lambda v.\ e^*_{while} \ = \ & \mathbf{if}\ e\ \mathbf{then}\ e'; (\lambda v.\ e^*_{while})\,\langle\rangle\ \mathbf{else\ skip} \\ & \mathbf{if}\ e\ \mathbf{then}\ e'; \mathbf{while}\ e\ \mathbf{do}\ e'\ \mathbf{else\ skip} \end{aligned}$$

Finalmente, bajo la hipótesis $\sigma, e \Rightarrow \mathbf{false}, \sigma'$, tenemos que

$$\sigma, (e^*_{while}/v \to \langle\rangle) \Rightarrow \langle\rangle, \sigma',$$

lo que prueba la primera regla. Por otr lado, bajo la hipótesis $\sigma, e \Rightarrow \mathbf{false}, \sigma'$, tenmos

$$\sigma, (e^*_{while}/v \to \langle\rangle) \Rightarrow z', \hat{\sigma},$$

producto de la hipótesis $e'; \mathbf{while}\ e\ \mathbf{do}\ e', \sigma' \Rightarrow z', \hat{\sigma}$