1 El lenguaje Iswim

Este lenguaje fue introducido por Peter Landing, 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 adición de 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:

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

La expresión **ref** e extiende el estado generando una referencia 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 el valor de la expresión e'.

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

Asumimos que Rf es un conjunto infinito de referencias, no es necesario precisar con detalle qué son las referencias, aunque si quisiéramos podríamos asumir que las mismas son, por ejemplo, números naturales. El conjunto de todos los estados se sigue escribiendo Σ, pero ahora los estados son funciones parciales de Rf en V. Los estados no mapean como antes identificadores en enteros, sino referencias en valores de cualquier tipo. Esto significa que la memoria es capaz de alojar cualquier tipo de valor (números, valores lógicos, funciones, tuplas, referencias). Dijimos que los estados son funciones parciales: es más, si $\sigma \in \Sigma$, entonces $dom(\sigma) \subset Rf$ y $dom(\sigma)$ es finito. Asumimos que existe una función new que aplicada a un estado devuelve una referencia nueva: $new(\sigma) \in Rf$ y $new(\sigma) \notin dom(\sigma)$. Para el lenguaje Iswim el conjunto de valores se extiende de la siguiente manera:

$$V = V_{int} + V_{bool} + V_{fun} + V_{tuple} + V_{ref}$$

$$D = (\Sigma \times V + \{\textbf{error}, \textbf{typeerror}\})_{\perp}$$

La función ι_{norm} aceptará ahora un par estado-valor: $\iota_{norm} <\sigma, z> \in D$. Como en el lenguaje aplicativo:

$Env = \langle var \rangle \to V$

$V_{int} = Z$

$V_{bool} = B$

$V_{fun} = \Sigma \times V \to D$

$V_{tuple} = V^*$

$V_{ref} = Rf$

$\iota_{int} \in V_{int} \to V$

$\iota_{bool} \in V_{bool} \to V$

$\iota_{fun} \in V_{fun} \to V$

$\iota_{tuple} \in V_{tuple} \to V$

$\iota_{ref} \in V_{ref} \to V$

Las funciones ahora deben reflejar la posibilidad de cambio del estado, es decir, una función no sólo puede utilizar el valor de su argumento, sino que los cómputos que realiza se efectúan en el estado resultante luego de evaluarse dicho argumento.

Disponemos además de resultados $err, tyerr, \bot \in D$ que constituyen la denotación de los errores y de la no-terminación.

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

$f_*\ \iota_{norm}\ \langle \sigma, z \rangle = f\ \langle \sigma, z \rangle$

$f_*\ err = err$

$f_*\ tyerr = tyerr$

$f_*\ \bot = \bot$

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

$f_{int}\ \langle \sigma, \iota_{int} i \rangle = f\ \langle \sigma, i \rangle$

$f_{int}\ \langle \sigma, \iota_\theta z \rangle = tyerr \qquad (\theta \neq int)$

De manera similar se definen f_{bool}, f_{fun}, f_{tuple}, f_{ref}.

La función semántica será de tipo:

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

La semántica de los construcciones típicamente imperativas es la siguiente:

$$[\![\mathbf{val}\ e]\!]\eta\sigma = (\lambda\ \langle \sigma', r \rangle \in \Sigma \times V_{ref}.\ \begin{cases} \iota_{norm}\ \langle \sigma', \sigma' r \rangle & r \in dom(\sigma') \\ err & c.c. \end{cases})_{ref*}([\![e]\!]\eta\sigma)$$

$[\![\mathbf{ref}\ e]\!]\eta\sigma = (\lambda\ \langle \sigma', z \rangle \in \Sigma \times V.\ \iota_{norm}\ \langle [\sigma' | r : z], \iota_{ref} r \rangle)_*([\![e]\!]\eta\sigma)$
donde r = new(σ')

$$[\![e := e']\!]\eta\sigma = (\lambda\,\langle\sigma', r\rangle \in \Sigma \times V_{ref}.\ (\lambda\,\langle\sigma'', z\rangle \in \Sigma \times V.\ \iota_{norm}\,\langle[\sigma''|r : z], z\rangle)_* \\ ([\![e']\!]\eta\sigma'))_{ref*}([\![e]\!]\eta\sigma)$$

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

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.

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

Lo mismo ocurre con la negación lógica:

$$[\![\neg e]\!]\eta\sigma = (\lambda\,\langle\sigma', b\rangle \in \Sigma \times V_{bool}.\ \iota_{norm}\,\langle\sigma', \iota_{bool}\neg b\rangle)_{bool*}([\![e]\!]\eta\sigma)$$

Un recurso similar sirve para los operadores binarios (salvo división y módulo):

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

Los operadores típicos del cálculo lambda se adaptan como sigue:

$$[\![ee']\!]\eta\sigma = (\lambda\,\langle\sigma', f\rangle \in \Sigma \times V_{fun}.\ f_*([\![e']\!]\eta\sigma'))_{fun*}([\![e]\!]\eta\sigma)$$

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

Finalmente, la semántica del letrec se define como sigue:

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

donde

$$f = \mathbf{Y}_{V_{fun}} F$$

$$F\ f\ \ \langle\sigma', z\rangle = [\![e]\!][\eta|w : \iota_{fun} f|v : z]\sigma'$$

1.1 Semántica operacional de Iswim

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

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

El predicado de evaluación tendrá la siguiente forma, que refleja no sólo el cómputo de un valor sino además la modificación del estado:

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

con la salvedad de que ahora los estados son funciones parciales de referencias en formas canónicas (y no en elementos de V). Todas las reglas aplicativas del lenguaje eager se incorporan con la indicación explícita de cómo se transforma el estado. Por ejemplo la regla

$$\frac{e \Rightarrow \lambda v.\ e_0 \quad e' \Rightarrow z' \quad (e_0/v \to z') \Rightarrow z}{ee' \Rightarrow z}$$

se transforma en:

$$\frac{\sigma, e \Rightarrow \lambda v.\ e_0, \sigma' \quad \sigma', e' \Rightarrow z', \sigma'' \quad \sigma'', (e_0/v \to z') \Rightarrow z, \sigma'''}{ee', s \Rightarrow z, \sigma'''}$$

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:

$$\frac{\sigma, e \Rightarrow r, \sigma' \quad \sigma', e' \Rightarrow z, \sigma''}{\sigma, e := e' \Rightarrow z, [\sigma''|r : z]}$$

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

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

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

1.2 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:

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

La semántica operacional nos permite verificar el significado esperado:

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

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 Si $\sigma, e \Rightarrow z, \sigma'$ y $\sigma', e' \Rightarrow z', \sigma''$, entonces

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

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.

Ampliar el ambiente con una nueva variable que denote una referencia también puede ser expresado mediante un let:

$$\textbf{newvar}\ v = e\ \textbf{in}\ e' \ =_{def}\ \textbf{let}\ v = \textbf{ref}\ e\ \textbf{in}\ e'$$

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

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

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

Propiedad Si $[\![e]\!]\eta\sigma = \iota_{norm}\langle \sigma', z\rangle$, entonces

$$[\![\textbf{newvar}\ v = e\ \textbf{in}\ e']\!]\eta\sigma = [\![e']\!][\eta|v : \iota_{ref} r][\sigma'|r : z] \quad \text{donde } r = new(\sigma')$$

Por último, la iteración es un tipo especial de declaración de función recursiva:

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

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:

$$\frac{\sigma, e \Rightarrow \mathbf{false}, \sigma'}{\sigma, \mathbf{while}\ e\ \mathbf{do}\ e' \Rightarrow \langle\rangle, \sigma'} \qquad \frac{\sigma, e \Rightarrow \mathbf{true}, \sigma' \quad \sigma', e'; \mathbf{while}\ e\ \mathbf{do}\ e' \Rightarrow z', \sigma''}{\sigma, \mathbf{while}\ e\ \mathbf{do}\ e' \Rightarrow z', \sigma''}$$

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

$$(\lambda v.\ e^*_{while})\,\langle\rangle \qquad (1).$$

Planteamos entonces la regla de la aplicación:

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

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 otro lado, bajo la hipótesis $\sigma, e \Rightarrow \mathbf{true}, \sigma'$, tenemos

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

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