

Lenguajes y Compiladores

Lenguajes Iswim-like

2023

Estructura de la materia a grandes rasgos:

Primera Parte: Lenguaje imperativo

Segunda Parte: Lenguaje aplicativo puro, y lenguaje aplicativo con referencias y asignación

Ejes de contenidos de la segunda parte

Cálculo Lambda

Lenguajes Aplicativos puros

Un Lenguaje Aplicativo con referencias y asignación

Cálculo Lambda

Lenguajes Aplicativos puros

$$\begin{aligned} \langle \text{exp} \rangle ::= & \langle \text{var} \rangle \mid \langle \text{exp} \rangle \langle \text{exp} \rangle \mid \lambda \langle \text{var} \rangle . \langle \text{exp} \rangle \\ & \mid \text{letrec } \langle \text{var} \rangle \equiv \lambda \langle \text{var} \rangle . \langle \text{exp} \rangle \text{ in } \langle \text{exp} \rangle \\ & \mid \langle \text{natconst} \rangle \mid \langle \text{boolconst} \rangle \\ & \mid -\langle \text{exp} \rangle \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \mid \dots \\ & \mid \langle \text{exp} \rangle \geq \langle \text{exp} \rangle \mid \langle \text{exp} \rangle \leq \langle \text{exp} \rangle \mid \langle \text{exp} \rangle < \langle \text{exp} \rangle \mid \dots \\ & \mid \langle \text{exp} \rangle \wedge \langle \text{exp} \rangle \mid \langle \text{exp} \rangle \vee \langle \text{exp} \rangle \mid \neg \langle \text{exp} \rangle \\ & \mid \text{if } \langle \text{exp} \rangle \text{ then } \langle \text{exp} \rangle \text{ else } \langle \text{exp} \rangle \end{aligned}$$
$$\langle \text{natconst} \rangle ::= 0 \mid 1 \mid 2 \mid \dots$$
$$\langle \text{boolconst} \rangle ::= \text{true} \mid \text{false}$$

Un Lenguaje Aplicativo con referencias y asignación

Lenguaje(s) Iswim-like

En los '60 Peter Landin utilizaba el cálculo lambda para dar la semántica de distintos lenguajes. Además definió la máquina abstracta SECD para la evaluación de expresiones (y explicó como programar la máquina abstracta).

Introdujo la noción de estado para explicar la asignación. Casi textualmente: “El significado de una IAE tiene dos aspectos, el descriptivo tiene que ver con el valor de la expresión; el imperativo con el cambio del estado de la máquina al ejecutar la IAE”.

El lenguaje Iswim propone incorporar una componente imperativa a un lenguaje aplicativo eager mediante *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.

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

$$\begin{aligned} \langle exp \rangle ::= & \dots \\ & | \text{ref } \langle exp \rangle \\ & | \text{val } \langle exp \rangle \\ & | \langle exp \rangle := \langle exp \rangle \\ & | \langle exp \rangle =_{ref} \langle exp \rangle \end{aligned}$$

Mientras que para las formas canónicas agregamos *referencias*:

$$\langle cnf \rangle ::= \dots \mid V_{ref}$$

La evaluación será una combinación de evaluación de aplicativo con semántica de transiciones de imperativo.

Intuición de nuevas expresiones. Crear referencia.

La expresión `ref e` extiende el estado generando una locación nueva que aloja el valor producido por `e` (no hay restricciones para el valor que puede adquirir `e`)

Intuición de nuevas expresiones. Dereferencia.

La expresión `val e` estará definida cuando `e` produzca un valor de tipo referencia, y la expresión completa devolverá lo alojado en esa referencia.

Intuición de nuevas expresiones. Asignación.

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** (esto es arbitrario).

Intuición de nuevas expresiones. Comprobación de igualdad.

La expresión $e =_{ref} e'$ comprueba que si ambas expresiones producen (evalúan a) referencias r y r' , respectivamente, ambas sean la misma; es decir es una forma de chequear igualdad de punteros.

Fragmento imperativo: skip

Podemos *definir* construcciones del lenguaje imperativo como la iteración y la declaración de variables locales sin extender la sintaxis abstracta. El orden de evaluación eager permite obtenerlas como azúcar sintáctico.

$$\mathbf{skip} =_{def} \langle \rangle$$

Fragmento imperativo: secuencia

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:

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

Note que aquí es fundamental el orden de evaluación eager para que la secuencia de ejecución de los comandos involucrados se respete.

Fragmento imperativo: newvar

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

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

Agotado el alcance de la declaración local, la referencia al lugar de memoria eventualmente se pierde (pero el lugar de memoria sigue definido). La restauración de v está dada por la semántica de la secuencia.

Fragmento imperativo: while

La iteración es un tipo espacial de declaración de función:

while e **do** $e' =_{def}$ **let** $\text{rec } f = \lambda v. \text{if } e \text{ then } e'; f \ v$ **else skip** **in** $f \langle \rangle$

Aquí las variables f y v no deben ocurrir en e ni e' .

Suponemos la existencia de un conjunto infinito de locaciones, llamado V_{ref} , de memoria que siempre alojan valores.

El conjunto de estados Σ está formado por funciones parciales definidas en un subconjunto finito de V_{ref} .

$$\Sigma = \bigcup_{F \subset_{fin} V_{ref}} F \rightarrow \langle cnf \rangle$$

Para $F \subset_{fin} V_{ref}$, $new(F) \in V_{ref}$ nos provee de una referencia con la propiedad $new(F) \notin F$. Por simplicidad sea $new(\sigma) = new(dom(\sigma))$.

Formas Canónicas:

$$\langle cnf \rangle ::= \dots \mid V_{ref}$$

Semántica big-step (evaluación):

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

Se define a través de reglas (axiomáticamente)

Evaluación del fragmento aplicativo

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

Por ejemplo la regla

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

se transforma en:

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

Reglas para el fragmento imperativo

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

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

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

$$\frac{\sigma, e \Rightarrow r, \sigma' \quad \sigma', e' \Rightarrow r', \sigma''}{\sigma, e =_{\mathit{ref}} e' \Rightarrow [r = r'], \sigma''}$$

Dominios para Semántica denotacional

Para la semántica denotacional tenemos estados “denotacionales”:

$$\Sigma = \bigcup_{F \subset_{fin} V_{ref}} F \rightarrow V$$

$$D = (\Sigma \times V + \{\mathbf{error}, \mathbf{typeerror}\})_{\perp} \quad \text{donde}$$

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

Las funciones ahora deben reflejar la posibilidad de cambio del estado, es decir, una función no sólo toma el valor sino además el estado producido por la evaluación del operando. Por ello:

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

Como antes, los valores $err, tyerr \in D$ constituyen la denotación de los errores.

Si $f \in \Sigma \times V \rightarrow D$, entonces $f_* \in D \rightarrow D$ se define:

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

$$f_*(err) = err$$

$$f_*(tyerr) = tyerr$$

$$f_*(\perp) = \perp$$

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

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

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

Función semántica

La función semántica será de tipo:

$$\llbracket _ \rrbracket \in \langle \textit{exp} \rangle \rightarrow Env \rightarrow \Sigma \rightarrow D$$

La semántica de las construcciones típicamente imperativas es la siguiente:

$$\llbracket \mathbf{val} \ e \rrbracket_{\eta\sigma} = (\lambda \langle \sigma', r \rangle. \mathbf{if} \ r \in \mathit{dom}(\sigma') \\ \mathbf{then} \ \iota_{\mathit{norm}} \langle \sigma', \sigma' r \rangle \mathbf{else} \ \mathit{err})_{\mathit{ref}*}(\llbracket e \rrbracket_{\eta\sigma})$$

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

donde $(r_{\sigma'} = \mathit{new}(\sigma'))$

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

$$\llbracket e =_{ref} e' \rrbracket \eta \sigma = (\lambda \langle \sigma', r \rangle. (\lambda \langle \sigma'', z \rangle. \iota_{norm} \langle \sigma'', \iota_{bool} r = r' \rangle)_{ref*} (\llbracket e' \rrbracket \eta \sigma'))_{ref*} (\llbracket e \rrbracket \eta \sigma)$$

Ecuaciones semánticas: fragmento puro

$$\llbracket 0 \rrbracket \eta \sigma = \iota_{norm} \langle \sigma, \iota_{int} 0 \rangle$$

$$\llbracket \mathbf{true} \rrbracket \eta \sigma = \iota_{norm} \langle \sigma, \iota_{bool} T \rangle$$

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

$$\llbracket e + e' \rrbracket \eta \sigma = (\lambda \langle \sigma', i \rangle$$

$$(\lambda \langle \sigma'', j \rangle. \iota_{norm} \langle \sigma'', \iota_{int} i + j \rangle)_{int*} (\llbracket e' \rrbracket \eta \sigma'))_{int*} (\llbracket e \rrbracket \eta \sigma)$$

$$\llbracket v \rrbracket \eta \sigma = \iota_{norm} \langle \sigma, \eta v \rangle$$

$$\llbracket e e' \rrbracket \eta \sigma = (\lambda \langle \sigma', f \rangle. f_* (\llbracket e' \rrbracket \eta \sigma'))_{fun*} (\llbracket e \rrbracket \eta \sigma)$$

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

$$\llbracket \text{letrec } w = \lambda v. e \text{ in } e' \rrbracket \eta \sigma = \llbracket e' \rrbracket [\eta \mid w : \iota_{fun} g] \sigma$$

donde

$$\begin{aligned} g &= \mathbf{Y}_{V_{fun}} F \\ F f \langle \sigma', z \rangle &= \llbracket e \rrbracket [\eta \mid w : \iota_{fun} f \mid v : z] \sigma' \end{aligned}$$

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.

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

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

Lema $\llbracket e; e' \rrbracket \eta \sigma = (\lambda \langle \sigma', z \rangle. \llbracket e' \rrbracket \eta \sigma')_*(\llbracket e \rrbracket \eta \sigma)$