Lenguajes y Compiladores

2017

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

1. Cálculo Lambda
2. Lenguajes Aplicativos puros
3. Un Lenguaje Aplicativo con referencias y asignación

Lenguajes Aplicativos

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle var \rangle \mid \langle exp \rangle \langle exp \rangle \mid \lambda \langle var \rangle . \langle exp \rangle \\
& \mid & \langle natconst \rangle \mid \langle boolconst \rangle \\
& \mid & -\langle exp \rangle \mid \langle exp \rangle + \langle exp \rangle \ \ldots \\
& \mid & \langle exp \rangle \geq \langle exp \rangle \mid \ldots \mid \langle exp \rangle \wedge \langle exp \rangle \mid \langle exp \rangle \vee \langle exp \rangle \mid \neg \\
& \mid & \textbf{if } \langle exp \rangle \textbf{ then } \langle exp \rangle \textbf{ else } \langle exp \rangle \\
& \mid & \langle \langle exp \rangle , \ldots, \langle exp \rangle \rangle \\
& \mid & \langle exp \rangle . \langle natconst \rangle \\
& \mid & \textbf{letrec } v \equiv \lambda u.e_0 \textbf{ in } e \\
& \mid & \textbf{rec } e \\
& \mid & \textbf{error} \mid \textbf{typeerror} \\
\langle natconst \rangle & ::= & 0 \mid 1 \mid 2 \mid \ldots .
\end{array}
$$

Lenguaje(s) Iswim-like

Peter Landin escribió una serie de artículos en los '60 donde 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).

Para explicar la asignación de lenguajes imperativos la noción de estado (en el primer paper le alcanzaba con un entorno) y referencias. 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 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.

Lenguaje Iswim

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

$$
\begin{array}{lcl}
\langle exp \rangle & ::= & \ldots \\
& | & \textbf{ref}\ \langle exp \rangle \\
& | & \textbf{val}\ \langle exp \rangle \\
& | & \langle exp \rangle := \langle exp \rangle \\
& | & \langle exp \rangle =_{ref} \langle exp \rangle
\end{array}
$$

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)

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** (esto es arbitrario).

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

Note que típicas construcciones del lenguaje imperativo como la iteración y la declaración de variables locales no se incorporan a la sintaxis abstracta. El orden de evaluación eager permite obtenerlas como azucar sintáctico.

$$\textbf{skip} \;=_{def}\; \langle\rangle$$

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:

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

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:

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

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:

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

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

Noción de Estado en Iswim

Suponemos la existencia de un conjunto infinito de locaciones de memoria que siempre alojan un valor del predomino V (y que llamaremos V_{ref}).

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

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

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 $new(\sigma) = new(dom(\sigma))$.

Semántica operacional de Iswim

Formas Canónicas:

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

Semántica big-step (evaluación):

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

Se define a través de reglas (axiomáticamente)

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:

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

Reglas para el fragmento imperativo

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

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

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

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

Dominios para Semántica denotacional

$$D = (\Sigma \times V + \{\textbf{error}, \textbf{typeerror}\})_\perp \quad \text{donde}$$
$$V = V_{int} + V_{bool} + V_{fun} + V_{tuple} + V_{ref}$$

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:

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

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

Funciones auxiliares

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

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

$f_*\ err = err$

$f_*\ tyerr = tyerr$

$f_*\ \ \bot = \bot$

Funciones auxiliares

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

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

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

Función semántica

La función semántica será de tipo:

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

Ecuaciones semánticas

La semántica de los construcciones típicamente imperativas es la siguiente:

$[\![\textbf{val}\ e]\!]\eta\sigma =$

$(\lambda\,\langle\sigma', r\rangle\,.\ \textbf{if}\ r \in dom(\sigma')\ \textbf{then}\ \iota_{norm}\,\langle\sigma', \sigma' r\rangle\ \ \textbf{else}\ err)_{ref*}([\![e]\!]\eta\sigma)$

$[\![\textbf{ref}\ e]\!]\eta\sigma = (\lambda\,\langle\sigma', z\rangle\,.\ \iota_{norm}\,\langle[\sigma'|r_{\sigma'} : z], \iota_{ref} r_{\sigma'}\rangle)_*([\![e]\!]\eta\sigma)$

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

Ecuaciones semánticas

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

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

Ecuaciones semánticas: fragmento puro

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

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

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

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

Ecuaciones semánticas: operadores del cálculo lambda

$$[\![v]\!]\eta\sigma = \iota_{norm}\,\langle \sigma, \eta\ v\rangle$$

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

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

Ecuaciones semánticas: letrec

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

donde

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

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

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:

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

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.

Justificación de la semántica de $e_0; e_1$

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

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

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

Semántica de newvar

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

Regla y ecuación semántica resultante (deben ser probadas):

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

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

donde $[\![e]\!]\eta\sigma = \iota_{norm} \langle \sigma', z \rangle$ y $r = new(\sigma')$

Iteración

$\textbf{while } e \textbf{ do } e' \ =_{def}$

letrec $w = \lambda v.$ **if** e **then** $e'; w\ v$ **else skip** **in** $w\ \langle\rangle$

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

Semántica de la iteración

Reglas resultantes:

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

$$\frac{\sigma, e \Rightarrow \textbf{true}, \sigma' \quad \sigma', e'; \textbf{while } e \textbf{ do } e' \Rightarrow z', \sigma''}{\sigma, \textbf{while } e \textbf{ do } e' \Rightarrow z', \sigma''}$$