Lenguajes y Compiladores

2015

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

Lenguajes Aplicativos

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

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.

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 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:

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

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 **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 **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 **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).

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 obteneralas como azucar sintáctico.

Fragmento imperativo: skip

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

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:

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

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:

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

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 eventualemnte se pierde (pero el lugar de memoria sigue definido)

Fragmento imperativo: while

La iteración es un tipo espacial de declaración de función:

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

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}).

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}.

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}.

$$\Sigma = \bigcup_{\substack{F \subset V_{ref} \\ F \text{ finito}}} F \to V$$

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}.

$$\Sigma = \bigcup_{\substack{F \subset V_{ref} \\ F \text{ finito}}} F \to V$$

Dado $\sigma \in \Sigma$, asumimos que $new(\sigma)$ nos provee de una referencia con la propiedad $new(\sigma) \notin dom(\sigma)$

Semántica denotacional

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

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

Predominio V_{fun}

Las funciones ahora deben reflejar la posibilidad de cambio del estado, es decir, una función no sólo computa un valor sino además eventualmente modifica el estado. Definimos ahora:

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

Predominio V_{fun}

Las funciones ahora deben reflejar la posibilidad de cambio del estado, es decir, una función no sólo computa un valor sino además eventualmente modifica el estado. Definimos ahora:

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

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:

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

$f_*\ err = err$

$f_*\ tyerr = tyerr$

$f_*\ \perp = \perp$

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:

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

De manera similar se definen los operadores "sub bool", "sub fun", etc.

Función semánticas

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:

Ecuaciones semánticas

La semántica de los construcciones típicamente imperativas es la siguiente:

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

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

Ecuaciones semánticas

La semántica de los construcciones típicamente imperativas es la siguiente:

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

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

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

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

Ecuaciones semánticas

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

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

Ecuaciones semánticas: fragmento puro

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

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

Ecuaciones semánticas: fragmento puro

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

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

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

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

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

Ecuaciones semánticas: operadores del cálculo lambda

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

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

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

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

Ecuaciones semánticas: letrec

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

donde

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

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

Semántica operacional de Iswim

Formas Canónicas:

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

Semántica operacional de Iswim

Formas Canónicas:

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

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.

Reglas para el fragmento aplicativo puro

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

Por ejemplo la regla

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

Reglas para el fragmento aplicativo puro

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

Por ejemplo la regla

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

se transforma en:

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

Reglas para el fragmento imperativo

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

Reglas para el fragmento imperativo

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

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

Reglas para el fragmento imperativo

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

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

$$\frac{e, \sigma \Rightarrow r, \sigma'}{\textbf{val}\ e, s \Rightarrow \sigma r, \sigma'}$$

Reglas para el fragmento imperativo

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

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

$$\frac{e, \sigma \Rightarrow r, \sigma'}{\textbf{val}\ e, s \Rightarrow \sigma r, \sigma'}$$

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

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:

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

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:

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

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:

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

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

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

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):

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

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):

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

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

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

Iteración

while e **do** e' $=_{def}$

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

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

Semántica de la iteración

Reglas resultantes:

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

Semántica de la iteración

Reglas resultantes:

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

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