Máquinas Virtuales

Compilación a Bytecode

15 de octubre de 2020

CEK... eficiente?

La máquina CEK era un evaluador "eficiente" de expresiones PCF, pero todavía bastante alejada de un procesador real.

CEK... eficiente?

La máquina CEK era un evaluador "eficiente" de expresiones PCF, pero todavía bastante alejada de un procesador real.

- Analiza sintaxis abstracta (árboles): fácil desde Haskell... ¿pero en assembly? Hay que serializar el árbol
- Completamente ligada al lenguaje: no podemos reusarla.
- Dos etapas, matching en elementos de la stack

CEK... eficiente?

La máquina CEK era un evaluador "eficiente" de expresiones PCF, pero todavía bastante alejada de un procesador real.

- Analiza sintaxis abstracta (árboles): fácil desde Haskell... ¿pero en assembly? Hay que serializar el árbol
- Completamente ligada al lenguaje: no podemos reusarla.
- Dos etapas, matching en elementos de la stack

Buscamos una forma de evaluación más **directa**,
una secuencia de **instrucciones**.

Máquinas de pila

Las máquinas de pila ejecutan una secuencia de instrucciones, donde cada una tiene algún efecto en una **pila de valores**.

Máquinas de pila

Las máquinas de pila ejecutan una secuencia de instrucciones, donde cada una tiene algún efecto en una **pila de valores**. El "hola mundo" de máquinas de pila: expresiones aritméticas.

$$e ::= N \mid e + e \mid -e$$

Máquinas de pila

Las máquinas de pila ejecutan una secuencia de instrucciones, donde cada una tiene algún efecto en una **pila de valores**. El "hola mundo" de máquinas de pila: expresiones aritméticas.

$$e ::= N \mid e + e \mid -e$$

$$\begin{aligned} \mathcal{C}(N) &= \texttt{CONST}(N) \\ \mathcal{C}(e_1 + e_2) &= \mathcal{C}(e_1);\ \mathcal{C}(e_2);\ \texttt{ADD} \\ \mathcal{C}(-e) &= \mathcal{C}(e);\ \texttt{NEG} \end{aligned}$$

Máquinas de pila

Las máquinas de pila ejecutan una secuencia de instrucciones, donde cada una tiene algún efecto en una **pila de valores**. El "hola mundo" de máquinas de pila: expresiones aritméticas.

$$e ::= N \mid e + e \mid -e$$

$$\begin{aligned} \mathcal{C}(N) &= \texttt{CONST}(N) \\ \mathcal{C}(e_1 + e_2) &= \mathcal{C}(e_1);\ \mathcal{C}(e_2);\ \texttt{ADD} \\ \mathcal{C}(-e) &= \mathcal{C}(e);\ \texttt{NEG} \end{aligned}$$

$$\mathcal{C}(5 + ((-2) + 8)) = \texttt{CONST}(5);\ \texttt{CONST}(2);\ \texttt{NEG};\ \texttt{CONST}(8);\ \texttt{ADD};\ \texttt{ADD}$$

Máquinas de pila

Las máquinas de pila ejecutan una secuencia de instrucciones, donde cada una tiene algún efecto en una **pila de valores**. El "hola mundo" de máquinas de pila: expresiones aritméticas.

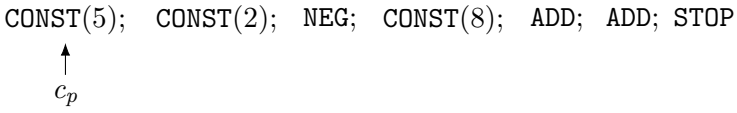
$$e ::= N \mid e + e \mid -e$$

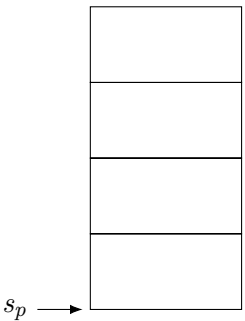
$$\begin{aligned} \mathcal{C}(N) &= \texttt{CONST}(N) \\ \mathcal{C}(e_1 + e_2) &= \mathcal{C}(e_1);\ \mathcal{C}(e_2);\ \texttt{ADD} \\ \mathcal{C}(-e) &= \mathcal{C}(e);\ \texttt{NEG} \end{aligned}$$

$$\mathcal{C}(5 + ((-2) + 8)) = \texttt{CONST}(5);\ \texttt{CONST}(2);\ \texttt{NEG};\ \texttt{CONST}(8);\ \texttt{ADD};\ \texttt{ADD}$$

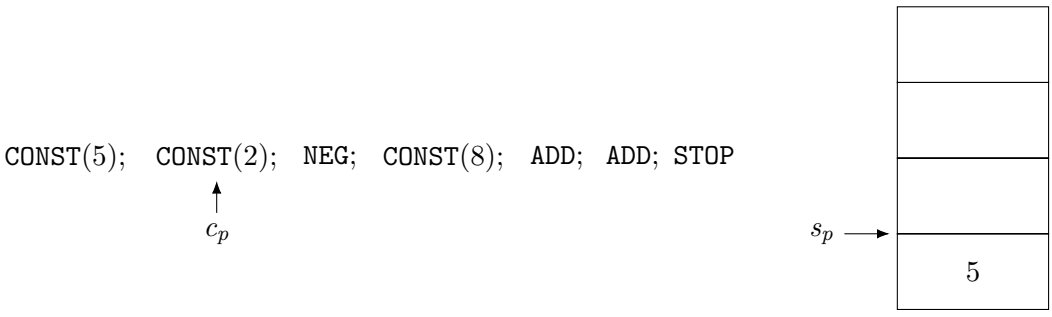
Esencialmente, notación polaca inversa.

Máquinas de pila — ejecución

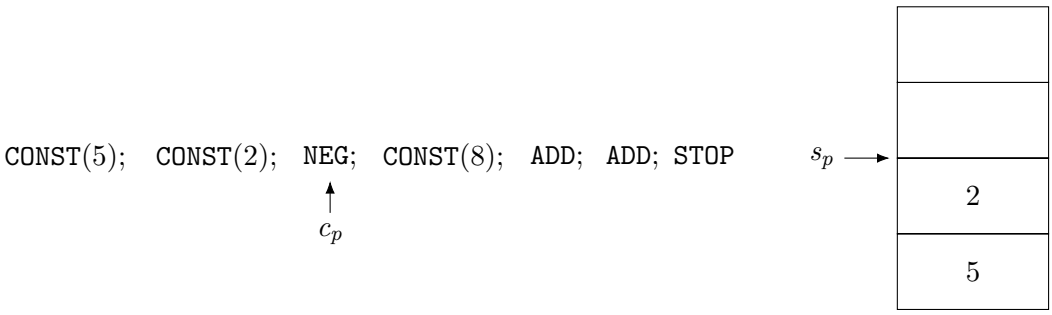




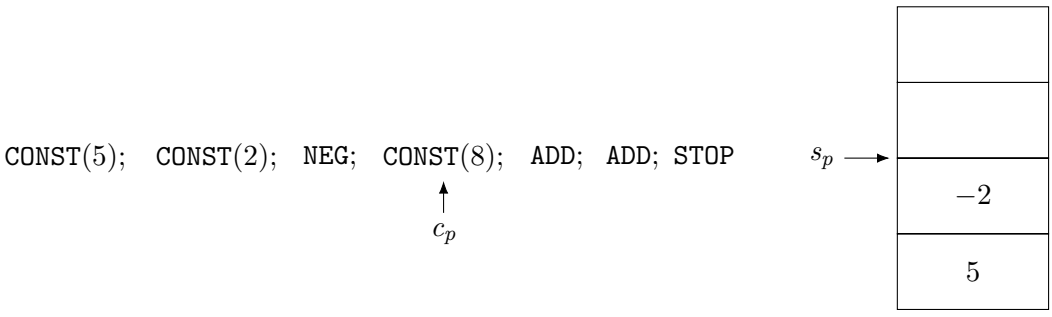
Máquinas de pila — ejecución



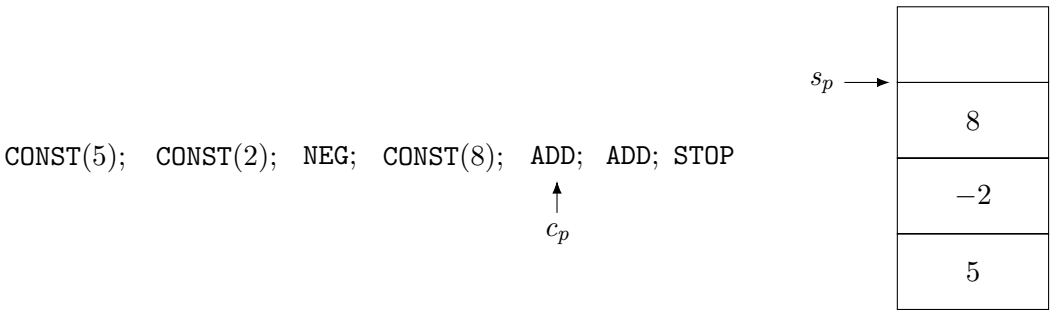
Máquinas de pila — ejecución



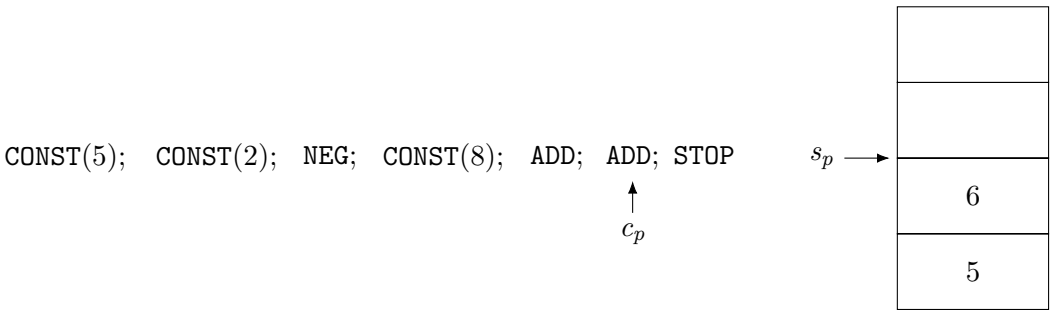
Máquinas de pila — ejecución



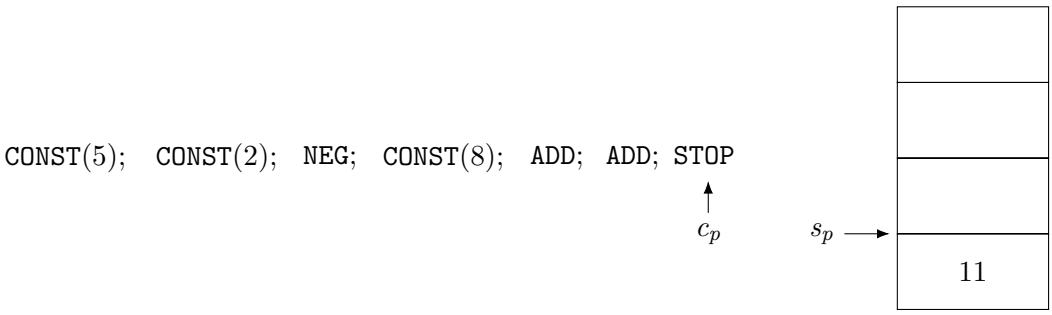
Máquinas de pila — ejecución



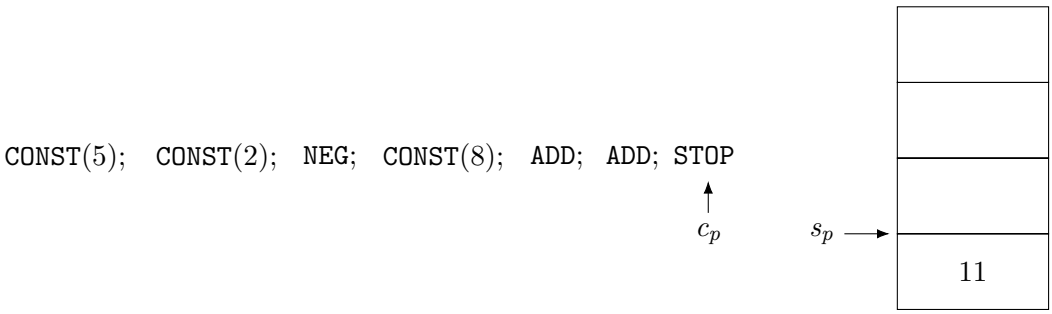
Máquinas de pila — ejecución



Máquinas de pila — ejecución



Máquinas de pila — ejecución



Vamos a hacer **exactamente lo mismo** para el fragmento aritmético de PCF.

Compilando el λ-cálculo — variables

Nuestra máquina de pila va a llevar también un **entorno** para las variables libres, similarmente a la CEK. La forma de un estado es $\langle c \mid e \mid s \rangle$.

Compilando el λ-cálculo — variables

Nuestra máquina de pila va a llevar también un **entorno** para las variables libres, similarmente a la CEK. La forma de un estado es $\langle c \mid e \mid s \rangle$. Es importante que c **es un puntero a código read-only**.

Compilando el λ-cálculo — variables

Nuestra máquina de pila va a llevar también un **entorno** para las variables libres, similarmente a la CEK. La forma de un estado es $\langle c \mid e \mid s \rangle$. Es importante que c **es un puntero a código read-only**.

$$\mathcal{C}(v_i) = \texttt{ACCESS}(i)$$

$$\langle \texttt{ACCESS}(i);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid e!i : s \rangle$$

Compilando el λ-cálculo — funciones

Las funciones tienen *instrucciones de retorno*

$$\begin{aligned} \mathcal{C}(\lambda t) &= \texttt{FUNCTION}(\mathcal{C}(t);\ \texttt{RETURN}) \\ \mathcal{C}(fe) &= \mathcal{C}(f);\ \mathcal{C}(e);\ \texttt{CALL} \end{aligned}$$

Compilando el λ-cálculo — funciones

Las funciones tienen *instrucciones de retorno*

$$\begin{aligned} \mathcal{C}(\lambda t) &= \texttt{FUNCTION}(\mathcal{C}(t);\ \texttt{RETURN}) \\ \mathcal{C}(fe) &= \mathcal{C}(f);\ \mathcal{C}(e);\ \texttt{CALL} \end{aligned}$$

las llamadas proveen las *direcciones de retorno*, usadas por `RETURN`.

$$\begin{array}{rlrcll} \langle \texttt{FUNCTION}(c_f); c \mid & e \mid & s \rangle & \longrightarrow & \langle c \mid & e \mid & (e, c_f) : s \rangle \\ \langle \texttt{CALL}; c \mid & e \mid & v : (e_f, c_f) : s \rangle & \longrightarrow & \langle c_f \mid & v : e_f \mid & (e, c)_{RA} : s \rangle \\ \langle \texttt{RETURN}; \_ \mid & \_ \mid & v : (e, c)_{RA} : s \rangle & \longrightarrow & \langle c \mid & e \mid & v : s \rangle \end{array}$$

Ejemplo

$$\mathcal{C}((\lambda x.\mathsf{succ}\ x)\ 10) =$$

Ejemplo

$$\mathcal{C}((\lambda x.\mathsf{succ}\ x)\ 10) = \texttt{FUNCTION}(\texttt{ACCESS}\ 0;\ \texttt{SUCC};\ \texttt{RETURN});\ \texttt{CONST}\ 10;\ \texttt{CALL}$$

Ejemplo

$$\mathcal{C}((\lambda x.\mathsf{succ}\ x)\ 10) = \texttt{FUNCTION}(\texttt{ACCESS}\ 0;\ \texttt{SUCC};\ \texttt{RETURN});\ \texttt{CONST}\ 10;\ \texttt{CALL}$$

Suponemos una continuación k...

$$\langle \texttt{FUNCTION}(B);\ \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid \quad e \mid \quad s \rangle \longrightarrow$$

Ejemplo

$$\mathcal{C}((\lambda x.\text{succ } x)\ 10) = \texttt{FUNCTION}(\texttt{ACCESS}\ 0;\ \texttt{SUCC};\ \texttt{RETURN});\ \texttt{CONST}\ 10;\ \texttt{CALL}$$

Suponemos una continuación k...

$$\begin{aligned} \langle \texttt{FUNCTION}(B);\ \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid \quad e \mid \quad s \rangle &\longrightarrow \\ \langle \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid \quad e \mid \quad (e, B) : s \rangle &\longrightarrow \end{aligned}$$

Ejemplo

$$\mathcal{C}((\lambda x.\text{succ } x)\ 10) = \texttt{FUNCTION}(\texttt{ACCESS}\ 0;\ \texttt{SUCC};\ \texttt{RETURN});\ \texttt{CONST}\ 10;\ \texttt{CALL}$$

Suponemos una continuación k...

$$\begin{aligned}
\langle \texttt{FUNCTION}(B);\ \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid \quad e \mid \quad s \rangle &\longrightarrow \\
\langle \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid \quad e \mid \quad (e, B) : s \rangle &\longrightarrow \\
\langle \texttt{CALL};\ k \mid \quad e \mid \quad 10 : (e, B) : s \rangle &\longrightarrow
\end{aligned}$$

Ejemplo

$$\mathcal{C}((\lambda x.\text{succ } x)\ 10) = \texttt{FUNCTION}(\texttt{ACCESS}\ 0;\ \texttt{SUCC};\ \texttt{RETURN});\ \texttt{CONST}\ 10;\ \texttt{CALL}$$

Suponemos una continuación k...

$$\begin{aligned}
\langle \texttt{FUNCTION}(B);\ \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid & \quad e \mid & s\rangle &\longrightarrow \\
\langle \texttt{CONST}\ 10;\ \texttt{CALL};\ k \mid & \quad e \mid & (e, B) : s\rangle &\longrightarrow \\
\langle \texttt{CALL};\ k \mid & \quad e \mid & 10 : (e, B) : s\rangle &\longrightarrow \\
\langle B \mid & \quad 10 : e \mid & (e, k)_{RA} : s\rangle &=
\end{aligned}$$

Ejemplo

$$\mathcal{C}((\lambda x.\text{succ } x)\ 10) = \texttt{FUNCTION}(\texttt{ACCESS } 0;\ \texttt{SUCC};\ \texttt{RETURN});\ \texttt{CONST } 10;\ \texttt{CALL}$$

Suponemos una continuación k...

$$\begin{aligned}
\langle \texttt{FUNCTION}(B);\ \texttt{CONST } 10;\ \texttt{CALL};\ k \mid \quad e \mid \quad s \rangle &\longrightarrow \\
\langle \texttt{CONST } 10;\ \texttt{CALL};\ k \mid \quad e \mid \quad (e, B) : s \rangle &\longrightarrow \\
\langle \texttt{CALL};\ k \mid \quad e \mid \quad 10 : (e, B) : s \rangle &\longrightarrow \\
\langle B \mid \quad 10 : e \mid \quad (e, k)_{RA} : s \rangle &= \\
\langle \texttt{ACCESS } 0;\ \texttt{SUCC};\ \texttt{RETURN} \mid \quad 10 : e \mid \quad (e, k)_{RA} : s \rangle &\longrightarrow
\end{aligned}$$

Ejemplo

$$\mathcal{C}((\lambda x.\text{succ } x)\ 10) = \texttt{FUNCTION}(\texttt{ACCESS } 0;\ \texttt{SUCC};\ \texttt{RETURN});\ \texttt{CONST } 10;\ \texttt{CALL}$$

Suponemos una continuación k...

$$\begin{aligned}
\langle \texttt{FUNCTION}(B);\ \texttt{CONST } 10;\ \texttt{CALL};\ k \mid & \quad e \mid & s\rangle &\longrightarrow \\
\langle \texttt{CONST } 10;\ \texttt{CALL};\ k \mid & \quad e \mid & (e, B) : s\rangle &\longrightarrow \\
\langle \texttt{CALL};\ k \mid & \quad e \mid & 10 : (e, B) : s\rangle &\longrightarrow \\
\langle B \mid & \quad 10 : e \mid & (e, k)_{RA} : s\rangle &= \\
\langle \texttt{ACCESS } 0;\ \texttt{SUCC};\ \texttt{RETURN} \mid & \quad 10 : e \mid & (e, k)_{RA} : s\rangle &\longrightarrow \\
\langle \texttt{SUCC};\ \texttt{RETURN} \mid & \quad 10 : e \mid & 10 : (e, k)_{RA} : s\rangle &\longrightarrow
\end{aligned}$$

Ejemplo

$$\mathcal{C}((\lambda x.\text{succ } x)\ 10) = \texttt{FUNCTION}(\texttt{ACCESS } 0;\ \texttt{SUCC};\ \texttt{RETURN});\ \texttt{CONST } 10;\ \texttt{CALL}$$

Suponemos una continuación k...

$$\begin{aligned}
\langle \texttt{FUNCTION}(B);\ \texttt{CONST } 10;\ \texttt{CALL};\ k \mid \quad & e \mid \quad s \rangle \quad \longrightarrow \\
\langle \texttt{CONST } 10;\ \texttt{CALL};\ k \mid \quad & e \mid \quad (e, B) : s \rangle \quad \longrightarrow \\
\langle \texttt{CALL};\ k \mid \quad & e \mid \quad 10 : (e, B) : s \rangle \quad \longrightarrow \\
\langle B \mid \quad & 10 : e \mid \quad (e, k)_{RA} : s \rangle \quad = \\
\langle \texttt{ACCESS } 0;\ \texttt{SUCC};\ \texttt{RETURN} \mid \quad & 10 : e \mid \quad (e, k)_{RA} : s \rangle \quad \longrightarrow \\
\langle \texttt{SUCC};\ \texttt{RETURN} \mid \quad & 10 : e \mid \quad 10 : (e, k)_{RA} : s \rangle \quad \longrightarrow \\
\langle \texttt{RETURN} \mid \quad & 10 : e \mid \quad 11 : (e, k)_{RA} : s \rangle \quad \longrightarrow
\end{aligned}$$

Ejemplo

$$\mathcal{C}((\lambda x.\text{succ } x)\ 10) = \texttt{FUNCTION}(\texttt{ACCESS } 0;\ \texttt{SUCC};\ \texttt{RETURN});\ \texttt{CONST } 10;\ \texttt{CALL}$$

Suponemos una continuación k...

$$\begin{aligned}
\langle \texttt{FUNCTION}(B);\ \texttt{CONST } 10;\ \texttt{CALL};\ k \mid \quad e \mid \quad s \rangle &\longrightarrow \\
\langle \texttt{CONST } 10;\ \texttt{CALL};\ k \mid \quad e \mid \quad (e, B) : s \rangle &\longrightarrow \\
\langle \texttt{CALL};\ k \mid \quad e \mid \quad 10 : (e, B) : s \rangle &\longrightarrow \\
\langle B \mid \quad 10 : e \mid \quad (e, k)_{RA} : s \rangle &= \\
\langle \texttt{ACCESS } 0;\ \texttt{SUCC};\ \texttt{RETURN} \mid \quad 10 : e \mid \quad (e, k)_{RA} : s \rangle &\longrightarrow \\
\langle \texttt{SUCC};\ \texttt{RETURN} \mid \quad 10 : e \mid \quad 10 : (e, k)_{RA} : s \rangle &\longrightarrow \\
\langle \texttt{RETURN} \mid \quad 10 : e \mid \quad 11 : (e, k)_{RA} : s \rangle &\longrightarrow \\
\langle k \mid \quad e \mid \quad 11 : s \rangle &
\end{aligned}$$

Muy lindo pero... ¿Y los puntos fijos?

NO queremos llevar otro tipo de clausura como valor, pero tampoco podemos saber a priori si una f es recursiva.

Muy lindo pero... ¿Y los puntos fijos?

NO queremos llevar otro tipo de clausura como valor, pero tampoco podemos saber a priori si una f es recursiva.
¿Podemos convertir una clausura de fixpoint en una clausura normal?

$$\mathcal{C}(\text{fix}.e) = \texttt{FIXPOINT}(e;\ \texttt{RETURN})$$

Muy lindo pero... ¿Y los puntos fijos?

NO queremos llevar otro tipo de clausura como valor, pero tampoco podemos saber a priori si una f es recursiva.
¿Podemos convertir una clausura de fixpoint en una clausura normal?

$$\mathcal{C}(\text{fix}.e) = \texttt{FIXPOINT}(e;\ \texttt{RETURN})$$

$$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$$

Muy lindo pero... ¿Y los puntos fijos?

NO queremos llevar otro tipo de clausura como valor, pero tampoco podemos saber a priori si una f es recursiva.
¿Podemos convertir una clausura de fixpoint en una clausura normal?

$$\mathcal{C}(\text{fix}.e) = \texttt{FIXPOINT}(e;\ \texttt{RETURN})$$

$$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$$

Para algún e_{fix}...

Muy lindo pero... ¿Y los puntos fijos?

NO queremos llevar otro tipo de clausura como valor, pero tampoco podemos saber a priori si una f es recursiva.
¿Podemos convertir una clausura de fixpoint en una clausura normal?

$$\mathcal{C}(\text{fix}.e) = \texttt{FIXPOINT}(e;\ \texttt{RETURN})$$

$$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$$

Para algún e_{fix}...

$$e_{\text{fix}} = (?, \qquad\qquad f) : e$$

Muy lindo pero... ¿Y los puntos fijos?

NO queremos llevar otro tipo de clausura como valor, pero tampoco podemos saber a priori si una f es recursiva.
¿Podemos convertir una clausura de fixpoint en una clausura normal?

$$\mathcal{C}(\text{fix}.e) = \texttt{FIXPOINT}(e;\ \texttt{RETURN})$$

$$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$$

Para algún e_{fix}...

$$\begin{aligned} e_{\text{fix}} &= (?, && f) : e \\ &= ((?, && f) : e, f) : e \end{aligned}$$

Muy lindo pero... ¿Y los puntos fijos?

NO queremos llevar otro tipo de clausura como valor, pero tampoco podemos saber a priori si una f es recursiva.
¿Podemos convertir una clausura de fixpoint en una clausura normal?

$$\mathcal{C}(\text{fix}.e) = \texttt{FIXPOINT}(e;\ \texttt{RETURN})$$

$$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$$

Para algún e_{fix}...

$$\begin{aligned} e_{\text{fix}} &= (?, & f) : e \\ &= ((?, & f) : e, f) : e \\ &= (((?, f) : e, f) : e, f) : e \end{aligned}$$

Muy lindo pero... ¿Y los puntos fijos?

NO queremos llevar otro tipo de clausura como valor, pero tampoco podemos saber a priori si una f es recursiva.
¿Podemos convertir una clausura de fixpoint en una clausura normal?

$$\mathcal{C}(\text{fix}.e) = \texttt{FIXPOINT}(e;\ \texttt{RETURN})$$

$$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$$

Para algún e_{fix}...

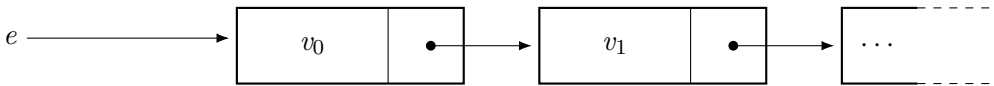
$$\begin{aligned} e_{\text{fix}} &= (?, f) : e \\ &= ((?, f) : e, f) : e \\ &= (((?, f) : e, f) : e, f) : e \end{aligned}$$

¿Y si...?

$$e_{\text{fix}} = (e_{\text{fix}}, f) : e$$

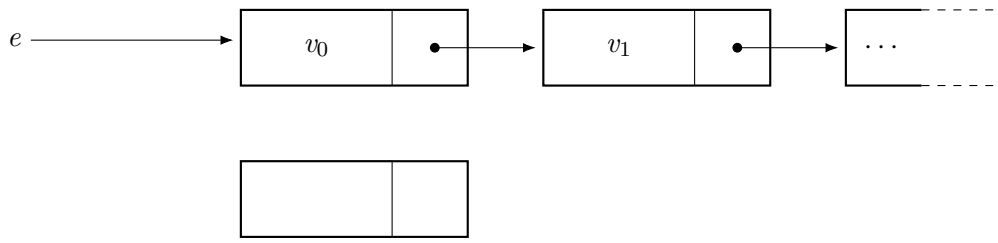
La verdad del punto fijo

$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$



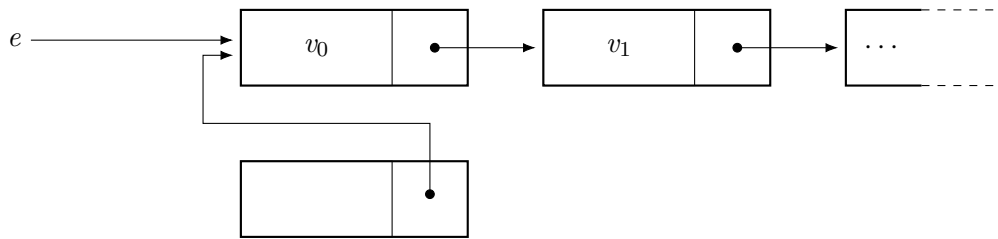
La verdad del punto fijo

$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$



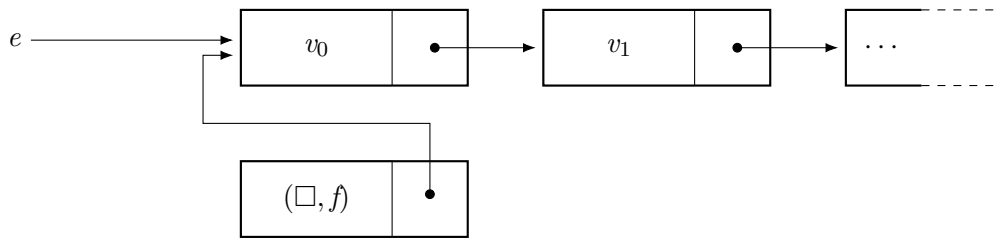
La verdad del punto fijo

$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$



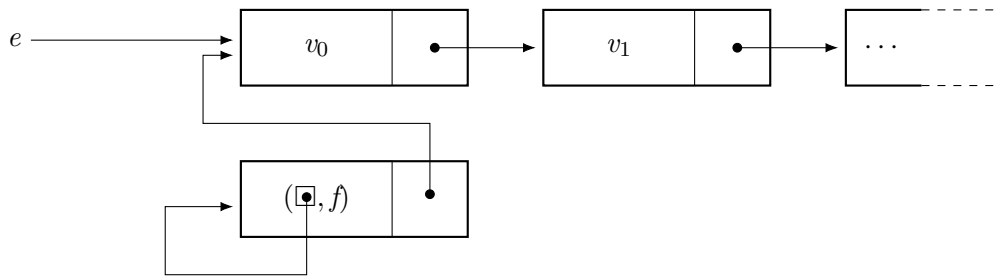
La verdad del punto fijo

$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$



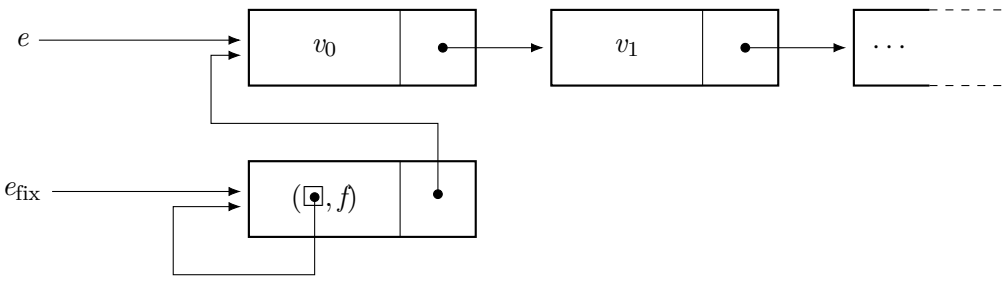
La verdad del punto fijo

$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$



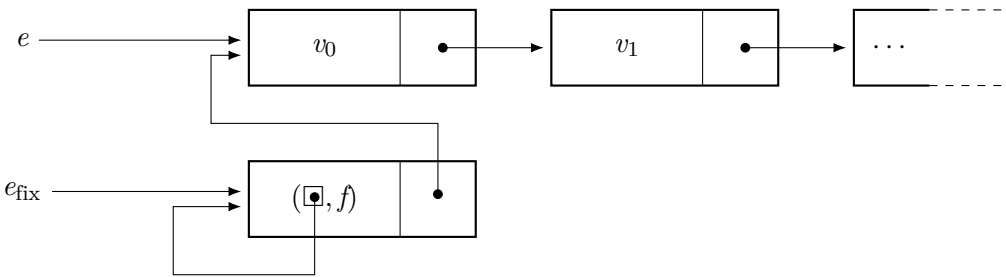
La verdad del punto fijo

$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$



La verdad del punto fijo

$\langle \texttt{FIXPOINT}(f);\ c \mid e \mid s \rangle \longrightarrow \langle c \mid e \mid (e_{\text{fix}}, f) : s \rangle$



Es una *estructura recursiva* (como `let ones = 1:ones` en anabólicos).

Serializando de verdad

El bytecode tiene que ser una cadena de enteros, nada más. Para compilar los nodos `FUNCTION`, debemos dejar información para **saltar** el cuerpo.

Serializando de verdad

El bytecode tiene que ser una cadena de enteros, nada más. Para compilar los nodos `FUNCTION`, debemos dejar información para **saltar** el cuerpo.

$$\begin{array}{rcl} e & \rightsquigarrow & [10, 20, 30, 40] \\ \texttt{FUNCTION}(e) & \rightsquigarrow & [0x42, 4, 10, 20, 30, 40] \end{array}$$

Serializando de verdad

El bytecode tiene que ser una cadena de enteros, nada más. Para compilar los nodos `FUNCTION`, debemos dejar información para **saltar** el cuerpo.

$$\begin{array}{rcl} e & \rightsquigarrow & [10, 20, 30, 40] \\ \texttt{FUNCTION}(e) & \rightsquigarrow & [0x42, 4, 10, 20, 30, 40] \end{array}$$

La máquina consume el "opcode", luego la longitud, y salta lo necesario hacia adelante.

Compilación de archivos

Esencialmente traducir:

$$\begin{array}{l} \texttt{let } v_1 = e_1 \\ \texttt{let } v_2 = e_2 \\ \ldots \\ \texttt{let } v_n = e_n \end{array} \longrightarrow \begin{array}{l} \texttt{let } v_1 = e_1 \texttt{ in} \\ \quad \texttt{let } v_2 = e_2 \texttt{ in} \\ \quad\quad \ldots \\ \quad\quad\quad \texttt{let } v_n = e_n \texttt{ in} \\ \quad\quad\quad\quad v_n \end{array}$$

compilar ese término, correr e imprimir el resultado.