Script generated by TTT

Title: Seidl: Virtual_Machines (26.06.2012)

Date: Tue Jun 26 14:02:39 CEST 2012

Duration: 88:18 min

Pages: 28

Example: $\textbf{let } a = 17 \textbf{ in let } f = \textbf{fun } b \to a + b \textbf{ in } f\ 42$

Disentanglement of the jumps produces:

0	loadc 17	2	mark B	3	B:	slide 2	1	pushloc 1
1	mkbasic	5	loadc 42	1		halt	2	eval
1	pushloc 0	6	mkbasic	0	A:	targ 1	2	getbasic
2	mkvec 1	6	pushloc 4	0		pushglob 0	2	add
2	mkfunval A	7	eval	1		eval	1	mkbasic
		7	apply	1		getbasic	1	return 1

Example: $\textbf{let } a = 17 \textbf{ in let } f = \textbf{fun } b \to a + b \textbf{ in } f\ 42$

Disentanglement of the jumps produces:

0	loadc 17	2	mark B	3	B:	slide 2	1	pushloc 1
1	mkbasic	5	loadc 42	1		halt	2	eval
1	pushloc 0	6	mkbasic	0	A:	targ 1	2	getbasic
2	mkvec 1	6	pushloc 4	0		pushglob 0	2	add
2	mkfunval A	7	eval	1		eval	1	mkbasic
		7	apply	1		getbasic	1	return 1

191

24 Structured Data

In the following, we extend our functional programming language by some datatypes.

24.1 Tuples

Constructors: $(.,\ldots,.)$, k-ary with $k \geq 0$;

Destructors: $\#j$ for $j \in \mathbb{N}_0$ (Projections)

We extend the syntax of expressions correspondingly:

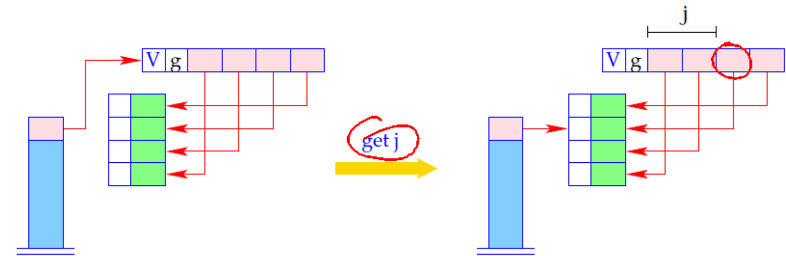
$$\begin{array}{lll} e & ::= & \ldots \mid (e_0,\ldots,e_{k-1}) \mid \#j\ e \\ & & \mid \textbf{let } (x_0,\ldots,x_{k-1}) = e_1 \textbf{ in } e_0 \end{array}$$

- In order to construct a tuple, we collect sequence of references on the stack. Then we construct a vector of these references in the heap using mkvec
- For returning components we use an indexed access into the tuple.

$$\begin{array}{lll} \text{code}_V\ (e_0, \ldots, e_{k-1})\ \rho\ \text{sd} & = & \text{code}_C\ e_0\ \rho\ \text{sd} \\ & & \text{code}_C\ e_1\ \rho\ (\text{sd}+1) \\ & & \ldots \\ & & \text{code}_C\ e_{k-1}\ \rho\ (\text{sd}+k-1) \\ & & \text{mkvec k} \end{array}$$

$$\begin{array}{lll} \text{code}_V\ (\#j\ e)\ \rho\ \text{sd} & = & \text{code}_V\ e\ \rho\ \text{sd} \\ & & \text{get j} \\ & & \text{eval} \end{array}$$

In the case of CBV, we directly compute the values of the e_i.




```
if (S[SP] == (V,g,v))
   S[SP] = v[j];
else Error "Vector expected!";
```


Inversion: Accessing all components of a tuple simulataneously:

$$e \equiv \textbf{let}\ (y_0, \ldots, y_{k-1}) = e_1\ \textbf{in}\ e_0$$

This is translated as follows:

$$\begin{array}{lll} \text{code}_V\ e\ \rho\ \text{sd} & = & \text{code}_V\ e_1\ \rho\ \text{sd} \\ & & \text{getvec k} \\ & & \text{code}_V\ e_0\ \rho'\ (\text{sd}+k) \\ & & \text{slide k} \end{array}$$

where $\rho' = \rho \oplus \{y_i \mapsto (L, sd+i+1) \mid i = 0, \ldots, k-1\}$.

The instruction getvec k pushes the components of a vector of length k onto the stack:

24.2 Lists

Lists are constructed by the constructors:

[] "Nil", the empty list;

"::" "Cons", right-associative, takes an element and a list.

Access to list components is possible by **match**-expressions ...

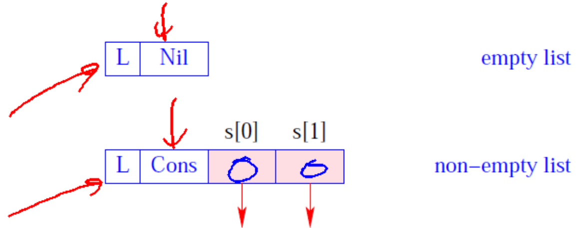
Example: The append function app:

$$\begin{array}{lll} \text{app} & = & \textbf{fun}\ l\ y \rightarrow \textbf{match}\ l\ \textbf{with} \\ & & \quad [] \quad \rightarrow \quad y \mid \\ & & \quad h :: t \quad \rightarrow \quad h :: (\text{app}\ t\ y) \end{array}$$

accordingly, we extend the syntax of expressions:

$$\begin{array}{lll} e & ::= & \ldots \mid [] \mid (e_1 :: e_2) \\ & & \mid (\textbf{match } e_0 \textbf{ with } [] \rightarrow e_1 \mid h :: t \rightarrow e_2) \end{array}$$

Additionally, we need new heap objects:



24.3 Building Lists

The new instructions nil and cons are introduced for building list nodes.

We translate for CBN:

$$\begin{array}{lll} \text{code}_V\ []\ \rho\ \text{sd} & = & \text{nil} \\ \text{code}_V\ (e_1 :: e_2)\ \rho\ \text{sd} & = & \text{code}_C\ e_1\ \rho\ \text{sd} \\ & & \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\ & & \text{cons} \end{array}$$

Note:

- With CBN: Closures are constructed for the arguments of ":";
- With CBV: Arguments of ":" are evaluated :-)

24.3 Building Lists

The new instructions nil and cons are introduced for building list nodes.

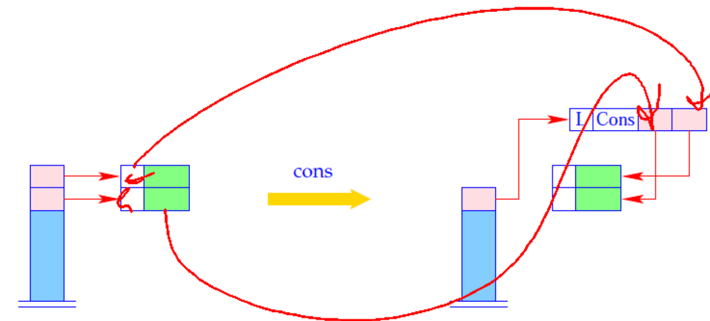
We translate for CBN:

$$\begin{array}{lll} \text{code}_V\ []\ \rho\ \text{sd} & = & \text{nil} \\ \text{code}_V\ (e_1 :: e_2)\ \rho\ \text{sd} & = & \text{code}_C\ e_1\ \rho\ \text{sd} \\ & & \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\ & & \text{cons} \end{array}$$

Note:

- With CBN: Closures are constructed for the arguments of ":";
- With CBV: Arguments of ":" are evaluated :-)

199




```
S[SP-1] = new (L,Cons, S[SP-1], S[SP]);
SP- -;
```

24.3 Building Lists

The new instructions nil and cons are introduced for building list nodes.

We translate for CBN:

$$\begin{aligned}
\text{code}_V\ []\ \rho\ \text{sd} &= \text{nil} \\
\text{code}_V\ (e_1 :: e_2)\ \rho\ \text{sd} &= \text{code}_C\ e_1\ \rho\ \text{sd} \\
&\ \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\
&\ \text{cons}
\end{aligned}$$

Note:

- With CBN: Closures are constructed for the arguments of ":";
- With CBV: Arguments of ":" are evaluated :-)

24.4 Pattern Matching

Consider the expression $e \equiv \textbf{match}\ e_0\ \textbf{with}\ [\,] \rightarrow e_1 \mid h :: t \rightarrow e_2$.

Evaluation of e requires:

- evaluation of e_0;
- check, whether resulting value v is an L-object;
- if v is the empty list, evaluation of e_1 ...
- otherwise storing the two references of v on the stack and evaluation of e_2. This corresponds to binding h and t to the two components of v.

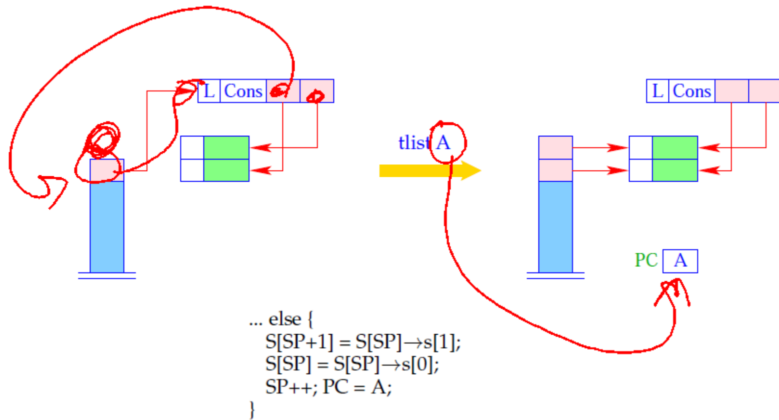
In consequence, we obtain (for CBN as for CBV):

$$\begin{aligned}
\text{code}_V\ e\ \rho\ \text{sd} = \quad & \text{code}_V\ e_0\ \rho\ \text{sd} \\
& \text{tlist A} \\
& \text{code}_V\ e_1\ \rho\ \text{sd} \\
& \text{jump B} \\
\text{A}: \quad & \text{code}_V\ e_2\ \rho'\ (\text{sd}+2) \\
& \text{slide 2} \\
\text{B}: \quad & \ldots
\end{aligned}$$

where $\rho' = \rho \oplus \{h \mapsto (L, sd+1), t \mapsto (L, sd+2)\}$.

The new instruction tlist A does the necessary checks and (in the case of Cons) allocates two new local variables:

```
h = S[SP];
if (H[h] != (L,...))
    Error "no list!";
if (H[h] == (_,Nil)) SP- -;
...
```



Example: The (disentangled) body of the function app with $\text{app} \mapsto (G, 0)$:

0		targ 2	3		pushglob 0	0	C:	mark D
0		pushloc 0	4		pushloc 2	3		pushglob 2
1		eval	5		pushloc 6	4		pushglob 1
1		tlist A	6		mkvec 3	5		pushglob 0
0		pushloc 1	4		mkclos C	6		eval
1		eval	4		cons	6		apply
1		jump B	3		slide 2	1	D:	update
2	A:	pushloc 1	1	B:	return 2			

Note:

Datatypes with more than two constructors need a generalization of the tlist instruction, corresponding to a `switch`-instruction :-)

In consequence, we obtain (for CBN as for CBV):

$$
\begin{array}{rcll}
\text{code}_V\ e\ \rho\ \text{sd} & = & & \text{code}_V\ e_0\ \rho\ \text{sd} \\
& & & \text{tlist A} \\
& & & \text{code}_V\ e_1\ \rho\ \text{sd} \\
& & & \text{jump B} \\
& & \text{A}: & \text{code}_V\ e_2\ \rho'\ (\text{sd}+2) \\
& & & \text{slide 2} \\
& & \text{B}: & \ldots
\end{array}
$$

where $\rho' = \rho \oplus \{h \mapsto (L, sd+1), t \mapsto (L, sd+2)\}$.

The new instruction tlist A does the necessary checks and (in the case of Cons) allocates two new local variables:

Example: The (disentangled) body of the function app with $\text{app} \mapsto (G, 0)$:

0		targ 2	3		pushglob 0	0	C:	mark D
0		pushloc 0	4		pushloc 2	3		pushglob 2
1		eval	5		pushloc 6	4		pushglob 1
1		tlist A	6		mkvec 3	5		pushglob 0
0		pushloc 1	4		mkclos C	6		eval
1		eval	4		cons	6		apply
1		jump B	3		slide 2	1	D:	update
2	A:	pushloc 1	1	B:	return 2			

Note:

Datatypes with more than two constructors need a generalization of the tlist instruction, corresponding to a `switch`-instruction :-)

24.5 Closures of Tuples and Lists

The general schema for code$_C$ can be optimized for tuples and lists:

$$\text{code}_C\ (e_0, \ldots, e_{k-1})\ \rho\ \text{sd} = \text{code}_V\ (e_0, \ldots, e_{k-1})\ \rho\ \text{sd} = \begin{array}{l} \text{code}_C\ e_0\ \rho\ \text{sd} \\ \text{code}_C\ e_1\ \rho\ (\text{sd}+1) \\ \ldots \\ \text{code}_C\ e_{k-1}\ \rho\ (\text{sd}+k-1) \\ \text{mkvec k} \end{array}$$

$$\text{code}_C\ []\ \rho\ \text{sd} = \text{code}_V\ []\ \rho\ \text{sd} = \text{nil}$$

$$\text{code}_C\ (e_1 : e_2)\ \rho\ \text{sd} = \text{code}_V\ (e_1 : e_2)\ \rho\ \text{sd} = \begin{array}{l} \text{code}_C\ e_1\ \rho\ \text{sd} \\ \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\ \text{cons} \end{array}$$

25 Last Calls

A function application is called last call in an expression e if this application could deliver the value for e.

A last call usually is the outermost application of a defining expression.

A function definition is called tail recursive if all recursive calls are last calls.

Examples:

$r\ t\ (h :: y)$ is a last call in **match** x **with** $[] \rightarrow y \mid h::t \rightarrow r\ t\ (h :: y)$

$f\ (x-1)$ is not a last call in **if** $x \leq 1$ **then** 1 **else** $x * f\ (x-1)$

Observation:

Last calls in a function body need no new stack frame!

Automatic transformation of tail recursion into loops!!!

The code for a last call $l \equiv (e'\ e_0 \ldots e_{m_1})$ inside a function f with k arguments must

1. allocate the arguments e_i and evaluate e' to a function (note: all this inside f's frame!);
2. deallocate the local variables and the k consumed arguments of f;
3. execute an apply.

$$\text{code}_V\ l\ \rho\ \text{sd} = \begin{array}{ll} \text{code}_C\ e_{m-1}\ \rho\ \text{sd} & \\ \text{code}_C\ e_{m-2}\ \rho\ (\text{sd}+1) & \\ \ldots & \\ \text{code}_C\ e_0\ \rho\ (\text{sd}+m-1) & \\ \text{code}_V\ e'\ \rho\ (\text{sd}+m) & \text{// Evaluation of the function} \\ \text{move}\ r\ (m+1) & \text{// Deallocation of } r \text{ cells} \\ \text{apply} & \end{array}$$

where $r = sd + k$ is the number of stack cells to deallocate.

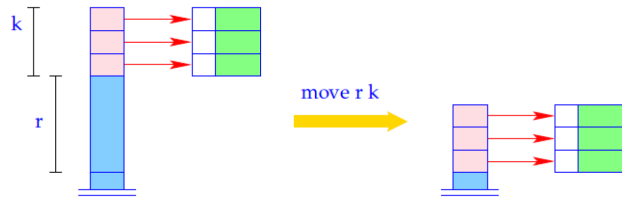
Example:

The body of the function

$r = \textbf{fun}\ x\ y \rightarrow \textbf{match}\ x\ \textbf{with}\ [] \rightarrow y \mid h::t \rightarrow r\ t\ (h :: y)$

0	targ 2	1		jump B	4		pushglob 0
0	pushloc 0				5		eval
1	eval	2	A:	pushloc 1	5		move 4 3
1	tlist A	3		pushloc 4			apply
0	pushloc 1	4		cons			
1	eval	3		pushloc 1	1	B:	return 2

Since the old stack frame is kept, return 2 will only be reached by the direct jump at the end of the []-alternative.




```
SP = SP – k – r;
for (i=1; i≤k; i++)
    S[SP+i] = S[SP+i+r];
SP = SP + k;
```


The Translation of Logic Languages

26 The Language Proll

Here, we just consider the core language Proll ("Prolog-light" :-). In particular, we omit:

- arithmetic;
- the cut operator;
- self-modification of programs through assert and retract.

Example:

$$
\begin{array}{lcl}
\mathsf{bigger}(X,Y) & \leftarrow & X = \mathit{elephant}, Y = \mathit{horse} \\
\mathsf{bigger}(X,Y) & \leftarrow & X = \mathit{horse}, Y = \mathit{donkey} \\
\mathsf{bigger}(X,Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{dog} \\
\mathsf{bigger}(X,Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{monkey} \\
\mathsf{is_bigger}(X,Y) & \leftarrow & \mathsf{bigger}(X,Y) \\
\mathsf{is_bigger}(X,Y) & \leftarrow & \mathsf{bigger}(X,Z), \mathsf{is_bigger}(Z,Y) \\
? \quad \mathsf{is_bigger}(\mathit{elephant}, \mathit{dog}) & &
\end{array}
$$