Script generated by TTT

Title: Seidl: Virtual_Machines (08.06.2015)

Date: Mon Jun 08 10:15:57 CEST 2015

Duration: 89:51 min

Pages: 26

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:

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

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

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

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:

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

196

- 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.

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

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

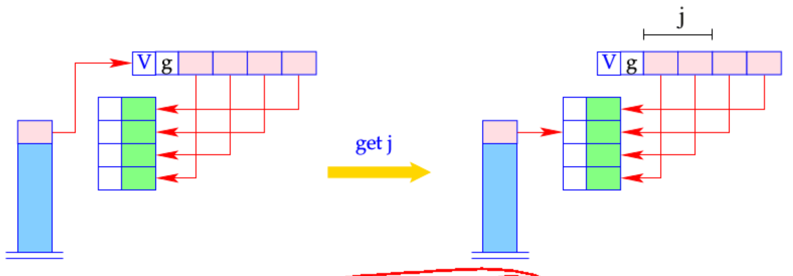
In the case of CBV, we directly compute the values of the e_i.

- 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.

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

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

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




```
if (S[SP] == (V,g,v)) if (j<g)
   S[SP] = v[j];
else Error "Vector index out of bounds!";
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:

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

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:

Inversion: Accessing all components of a tuple simulataneously:

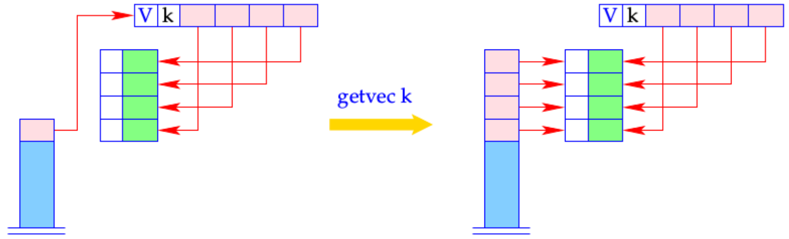
$$e \equiv \textbf{let } (y_0, \ldots, y_{k-1}) = e_1 \textbf{ in } e_0$$

This is translated as follows:

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

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:




```
if (S[SP] == (V,k,v)) {
   SP−;
   for(i=0; i<k; i++) {
      SP++; S[SP] = v[i];
   }
} 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:

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

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 ...

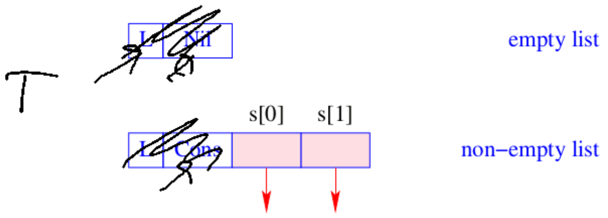
Example The append function app:

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

accordingly, we extend the syntax of expressions:

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

Additionally, we need new heap objects:



24.3 Building Lists

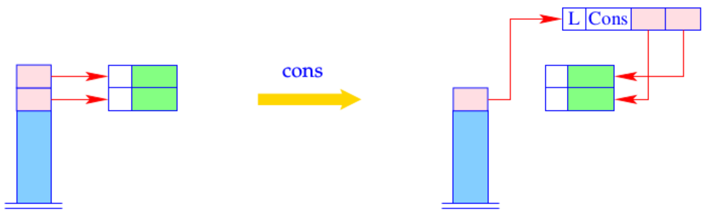
The new instructions nil and cons are introduced for building list nodes.

We translate for CBN:

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

Note:

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



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

24.4 Pattern Matching

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

Evaluation of e requires:

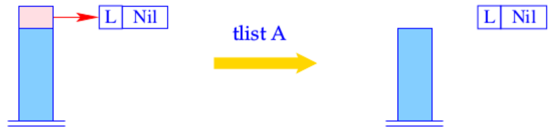
- evaluation of e_0;
- check, whether resulting value v is an L-object;
- if v is the empty list, evaluation of e_1 ...
- otherwise storing the two references of v on the stack and evaluation of e_2. This corresponds to binding h and t to the two components of v.

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

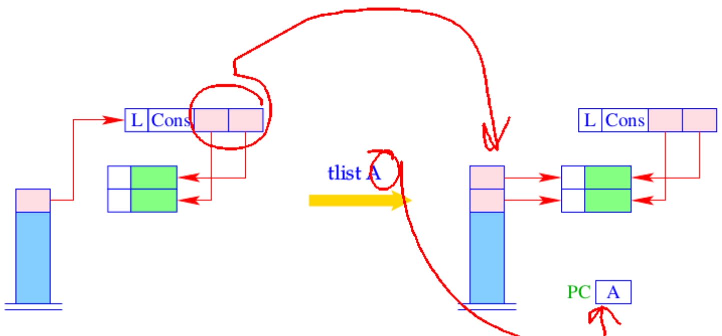
$$
\begin{array}{lll}
\text{code}_V\; e\; \rho\; \text{sd} & = & \text{code}_V\; e_0\; \rho\; \text{sd} \\
 & & \text{tlist A} \\
 & & \text{code}_V\; e_1\; \rho\; \text{sd} \\
 & & \text{jump B} \\
 & \text{A}: & \text{code}_V\; e_2\; \rho'\; (\text{sd}+2) \\
 & & \text{slide 2} \\
 & \text{B}: & \ldots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

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:




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


```
... else {
  S[SP+1] = S[SP]→s[1];
  S[SP] = S[SP]→s[0];
  SP++; PC = A;
}
```


Example The (disentangled) body of the function app with 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 :-)

24.2 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:

$$\text{app} \quad = \quad \textbf{fun } l\ y \to \textbf{match } l \textbf{ with} \quad \mid\ [\,] \to y \quad \mid\ h :: t \to h :: (\text{app}\ t\ y)$$

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:

$$\text{app} \quad = \quad \textbf{fun } l\ y \to \textbf{match } l \textbf{ with} \quad \mid\ [\,] \to y \quad \mid\ h :: t \to h :: (\text{app}\ t\ y)$$

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:

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

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

$$\text{code}_C\ (e_1 :: e_2)\ \rho\ \text{sd} = \text{code}_V\ (e_1 :: e_2)\ \rho\ \text{sd} = \text{code}_C\ e_1\ \rho\ \text{sd};\ \text{code}_C\ e_2\ \rho\ (\text{sd}+1);\ \text{cons}$$

24.5 Closures of Tuples and Lists

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

$$
\begin{aligned}
\text{code}_C\ (e_0,\ldots,e_{k-1})\ \rho\ \text{sd} &= \text{code}_V\ (e_0,\ldots,e_{k-1})\ \rho\ \text{sd} &=&\ \text{code}_C\ e_0\ \rho\ \text{sd} \\
&&&\ \text{code}_C\ e_1\ \rho\ (\text{sd}+1) \\
&&&\ \ldots \\
&&&\ \text{code}_C\ e_{k-1}\ \rho\ (\text{sd}+k-1) \\
&&&\ \text{mkvec k} \\
\text{code}_C\ []\ \rho\ \text{sd} &= \text{code}_V\ []\ \rho\ \text{sd} &=&\ \text{nil} \\
\text{code}_C\ (e_1 :: e_2)\ \rho\ \text{sd} &= \text{code}_V\ (e_1 :: e_2)\ \rho\ \text{sd} &=&\ \text{code}_C\ e_1\ \rho\ \text{sd} \\
&&&\ \text{code}_C\ e_2\ \rho\ (\text{sd}+1) \\
&&&\ \text{cons}
\end{aligned}
$$

25 Last Calls

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

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** $[]\ \to y \mid h :: t\ \to r\ t\ (h :: y)$
$f\ (x-1)$ 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!

$\Longrightarrow$

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.

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

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