Script generated by TTT

Title: Seidl: Virtual_Machines (09.06.2015)

Date: Tue Jun 09 10:15:43 CEST 2015

Duration: 91:24 min

Pages: 26

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.

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

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

$r \mapsto (G,0)$ $x \mapsto (L,0)$ $h \mapsto (L,1)$

$y \mapsto (L,-1)$ $t \mapsto (L,2)$

Example

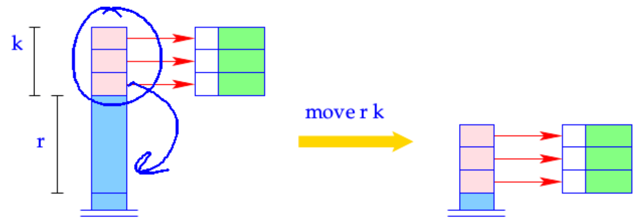
V-code for the body of the function

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

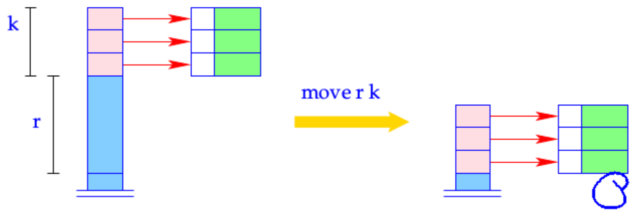
with CBN semantics:

0	targ 2	1		jump B	4		pushglob 0
0	pushloc 0				5		eval
1	eval	2	A:	$\boxed{\text{pushloc 1}}$	5		move 4 3
1	tlist A	3		$\boxed{\text{pushloc 4}}$			apply
0	pushloc 1	4		$\boxed{\text{cons}}$			~~slide 2~~
1	eval	3		$\boxed{\text{pushloc 1}}$	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;




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


26 Exceptions

Example

$$
\begin{array}{l}
\textbf{let rec } gcd = \textbf{fun } x\ y \rightarrow \\
\qquad \textbf{if } x \leq 0 \mid\mid y \leq 0 \textbf{ then raise } 0 \\
\qquad \textbf{else if } x = y \textbf{ then } x \\
\qquad \textbf{else if } y < x \textbf{ then } gcd\ (x-y)\ y \\
\qquad \textbf{else } gcd\ x\ (y-x) \\
\textbf{in try } gcd\ 0\ 5 \\
\textbf{with } z\ \rightarrow z
\end{array}
$$

For simplicity, we assume that all raised exception values are of any type.

For every try expression, we maintain:

- An exception frame on the stack, which contains all relevant information to handle the exception;
- The exception pointer XP, which points the the current exception frame.

Each exception frame must record

- the negative continuation address, i.e., the address of the code for the handler;
- the global pointer and
- the frame pointer; as well as
- the old exception pointer.

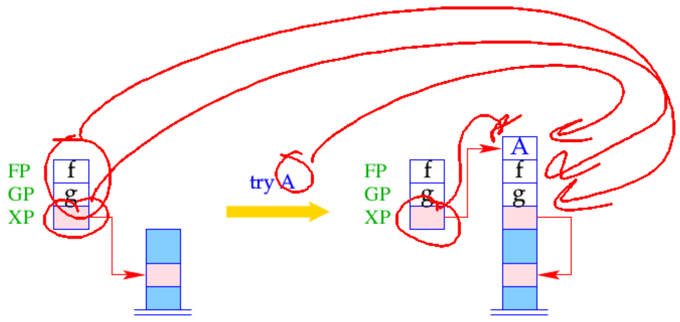
For an expression of the following form:

$$e \equiv \textbf{try } e_1 \textbf{ with } x \rightarrow e_2$$

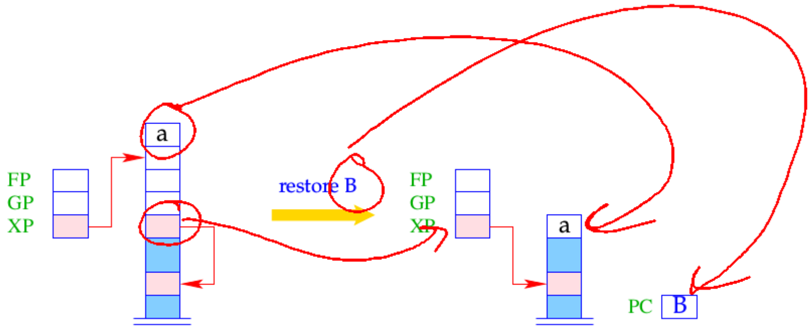
we generate:

$$
\begin{array}{rcrl}
\text{code}_V\ e\ \rho\ \text{sd} & = & & \text{try A} \\
& & & \text{code}_V\ e_1\ \rho\ (\text{sd}+4) \\
& & & \text{restore B} \\
& & A: & \text{code}_V\ e_2\ \rho'\ (\text{sd}+1) \\
& & & \text{slide 1} \\
& & B: & \ldots
\end{array}
$$

where $\quad \rho' = \rho \oplus \{x \mapsto (L, \text{sd}+1)\}.$



S[SP+1]=XP;
S[SP+2] = GP;
S[SP+3]=FP;
S[SP+4] = A;
SP =SP+4;
XP = SP;



PC = B;
XP = S[SP-4];
S[SP-4] = S[SP];
SP = SP-4;

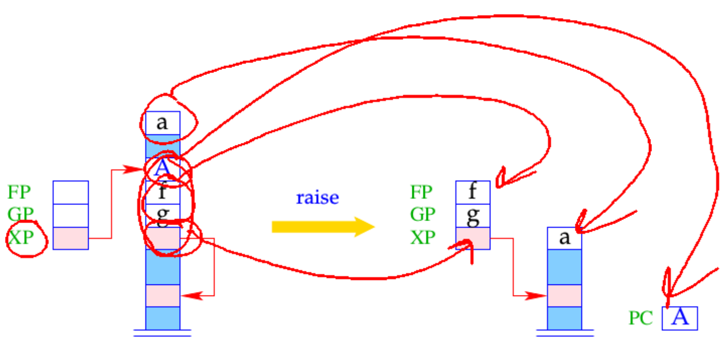
Now we have all provisions to raise exceptions.

For these, we do:

- We give up the current comutational context;
- We restore the context of the closest surrounding try expression;
- We hand over the exception value to the exception handler.

Thus, we translate:

$$
\begin{array}{lll}
\text{code}_V\ (\textbf{raise}\ e)\ \rho\ \text{sd} & = & \text{code}_V\ e\ \rho\ \text{sd} \\
 & & \text{raise}
\end{array}
$$



a = S[SP];
SP = XP-3;
PC = S[XP];
FP = S[XP-1];
GP = S[XP-2];
XP = S[XP-3];
S[SP] = a;

Example

The V-code for *gcd* is given by:

0		alloc 1	2	B:	rewrite 1	10		mkbasic
1		pushloc 0	1		try C	10		pushloc 9
2		mkvec 1	5		mark D	11		apply
2		mkfun A	8		loadc 5	6	D:	restore E
2		jump B	9		mkbasic	2	C:	pushloc 0
0	A:	targ 2	9		loadc 0	3		slide 1
		...				2	E:	slide 1
		return 2						

Remarks

- In Ocaml, exceptions may also be raised by the runtime system.
- Therefore, exceptions form a datatype on their own, which can be extended with further constructors by the programmer.
- The handler performs pattern matching on the exception value.
- If the given exception value is not matched, the exception value is raised again.

Remarks

- In Ocaml, exceptions may also be raised by the runtime system.
- Therefore, exceptions form a datatype on their own, which can be extended with further constructors by the programmer.
- The handler performs pattern matching on the exception value.
- If the given exception value is not matched, the exception value is raised again.

Caveat

Exceptions only make sense in CBV languages !!

Why??

Remarks

- In Ocaml, exceptions may also be raised by the runtime system.
- Therefore, exceptions form a datatype on their own, which can be extended with further constructors by the programmer.
- The handler performs pattern matching on the exception value.
- If the given exception value is not matched, the exception value is raised again.

Caveat

Exceptions only make sense in CBV languages !!

Why??

225

The Translation of Logic Languages

27 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

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

... in Concrete Syntax:

$$
\begin{array}{lcl}
\text{bigger}\mathit{elephant}, \mathit{horse}). & & \\
\text{bigger}(\mathit{horse}, \mathit{donkey}). & & \\
\text{bigger}(\mathit{donkey}, \mathit{dog}). & & \\
\text{bigger}(\mathit{donkey}, \mathit{monkey}). & & \\
\text{is\_bigger}(X,Y) & :- & \text{bigger}(X,Y). \\
\text{is\_bigger}(X,Y) & :- & \text{bigger}(X,Z), \text{is\_bigger}(Z,Y). \\
?- \text{is\_bigger}(\mathit{elephant}, \mathit{dog}). & &
\end{array}
$$

Example

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

A More Realistic Example

$$\begin{array}{lcl}
\text{app}(X,Y,Z) & \leftarrow & X = [\,], Y = Z \\
\text{app}(X,Y,Z) & \leftarrow & X = [H|X'], Z = [H|Z'], \text{app}(X',Y,Z') \\
? \;\; \text{app}(X, [Y,c], [a,b,Z]) & &
\end{array}$$

A More Realistic Example

$\text{app}([\,], Z, Z).$

$\text{app}([H|X'], Y, [H|Z']) \;\; :- \;\; \text{app}(X', Y, Z').$

$?- \;\; \text{app}(X, [Y,c], [a,b,Z]).$

A More Realistic Example

$$\begin{array}{lcl}
\boxed{\text{app}(X,Y,Z)} & \leftarrow & \boxed{X = [\,], Y = Z} \\
\boxed{\text{app}(X,Y,Z)} & \leftarrow & \boxed{X = [H|X'], Z = [H|Z']}\, \boxed{\text{app}(X',Y,Z')} \\
? \;\; \boxed{\text{app}(X, [Y,c], [a,b,Z])} & &
\end{array}$$

A More Realistic Example

$$\mathsf{app}(X, Y, Z) \;\leftarrow\; X = [\,],\; Y = Z$$
$$\mathsf{app}(X, Y, Z) \;\leftarrow\; X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z')$$
$$?\;\; \mathsf{app}(X, [Y, c], [a, b, Z])$$

$[a \mid [b]]$

Remark

$[\,]$	$==$	the atom empty list
$[H|Z]$	$==$	binary constructor application
$[a, b, Z]$	$==$	shortcut for: $[a|[b|[Z|[\,]]]]$

$\neq$

$[a|[b|Z]]$

A More Realistic Example

$$\mathsf{app}([\,], Z, Z).$$
$$\mathsf{app}([H|X'], Y, [H|Z']) \;:-\; \mathsf{app}(X', Y, Z').$$
$$?-\;\; \mathsf{app}(X, [Y, c], [a, b, Z]).$$

A More Realistic Example

$$\mathsf{app}(X, Y, Z) \;\leftarrow\; X = [\,],\; Y = Z$$
$$\mathsf{app}(X, Y, Z) \;\leftarrow\; X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z')$$
$$?\;\; \mathsf{app}(X, [Y, c], [a, b, Z])$$

Remark

$[\,]$	$==$	the atom empty list
$[H|Z]$	$==$	binary constructor application
$[a, b, Z]$	$==$	shortcut for: $[a|[b|[Z|[\,]]]]$

232