Script generated by TTT

Title: Seidl: Virtual Machines (20.05.2014)

Date: Tue May 20 10:15:10 CEST 2014

Duration: 91:29 min

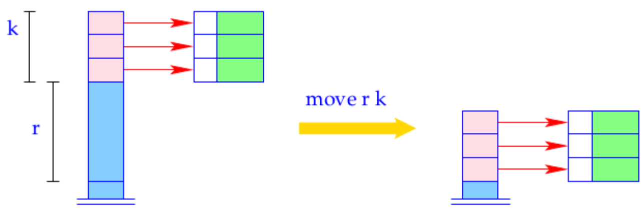
Pages: 32

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.



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

Example

$\rho = \{ r \mapsto (G, 0), \; x \mapsto (L, 0), \; y \mapsto (L, -1) \}$

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:	pushloc 1	5		move 4 3
1	tlist A	3		pushloc 4			apply
0	pushloc 1	4		cons			~~slide 2~~
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.

$h \mapsto (L, 1)$
$t \mapsto (L, 2)$

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:	pushloc 1	5		move 4 3
1		tlist A	3		pushloc 4			apply
0		pushloc 1	4		cons			slide 2
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.

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 e_2;
- the global pointer and
- the frame pointer; as well as
- the old exception pointer.

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 e_2;
- the global pointer and
- the frame pointer; as well as
- the old exception pointer.

26 Exceptions

Example

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

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 e_2;
- the global pointer and
- the frame pointer; as well as
- the old exception pointer.

217

For an expression of the following form:

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

we generate:

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

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

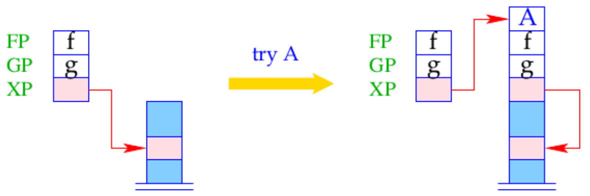
For an expression of the following form:

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

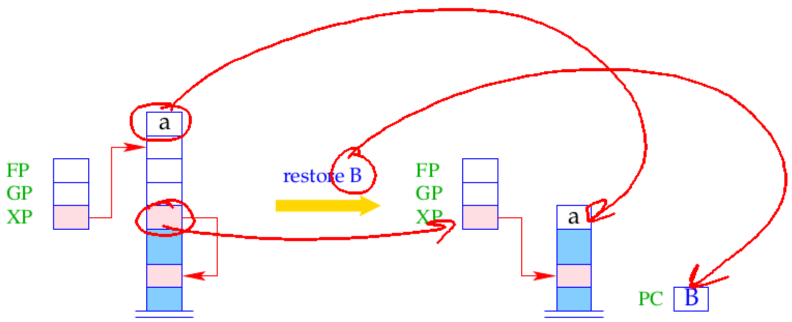
we generate:

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

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



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



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

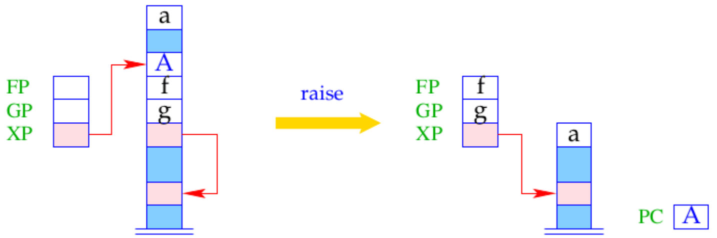
Now we have all provisions to raise exceptions.

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:

$$\begin{array}{rcl} \mathrm{code}_V\ (\textbf{raise}\ e)\ \rho\ \mathrm{sd} & = & \mathrm{code}_V\ e\ \rho\ \mathrm{sd} \\ & & \mathtt{raise} \end{array}$$

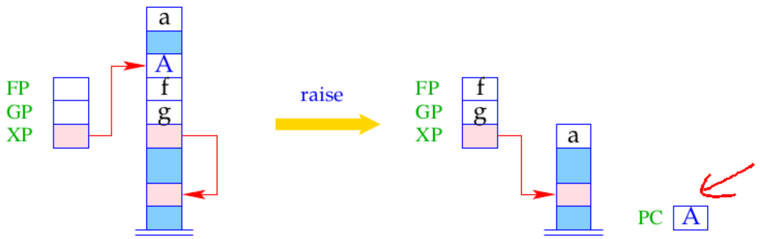


a = S[SP];
SP = XP-3;
PC = S[XP];
FP = S[XP-1];
GP = S[XP-2];
XP = S[XP-3];
S[SP] = a;

Example

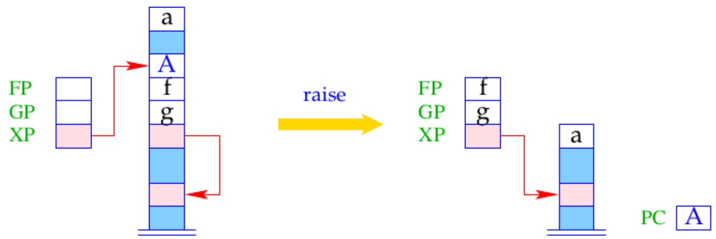
The code for gcd is given by:

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



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

$2 \mapsto (L, 2)$
$\text{gcd} \mapsto (C, 1)$



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

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 e_2;
- the global pointer and
- the frame pointer; as well as
- the old exception pointer.

Remarks

- In Ocaml, exceptions form a datatype on their own.
- 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 form a datatype on their own.
- 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??

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:

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

A More Realistic Example:

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

Example:

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

app([], Y, Y).

app([H|X], Y, [H|Z']) :-

app(X', Y, Z').

A More Realistic Example:

$$
\begin{array}{lll}
\mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\; Y = Z \\
\mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z')
\end{array}
$$

$? \quad \mathsf{app}(X, [Y, c], [a, b, Z])$

?- app(X, [Y, c], [a, b, Z]).

A More Realistic Example:

$$
\begin{array}{lll}
\mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\; Y = Z \\
\mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z')
\end{array}
$$

$? \quad \mathsf{app}(X, [Y, c], [a, b, Z])$

Remark:

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

A program p is constructed as follows:

$$
\begin{array}{lll}
t & ::= & a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\
g & ::= & p(t_1, \ldots, t_k) \mid X = t \\
c & ::= & p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\
p & ::= & c_1 \ldots . c_m ? g
\end{array}
$$

- A term t either is an atom, a variable, an anonymous variable or a constructor application.
- A goal g either is a literal, i.e., a predicate call, or a unification.
- A clause c consists of a head $p(X_1, \ldots, X_k)$ with predicate name and list of formal parameters together with a body, i.e., a sequence of goals.
- A program consists of a sequence of clauses together with a single goal as query.

A More Realistic Example:

$$
\begin{array}{lll}
\mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\; Y = Z \\
\mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\; Z = [H|Z'],\; \mathsf{app}(X', Y, Z')
\end{array}
$$

$? \quad \mathsf{app}(X, [Y, c], [a, b, Z])$

Remark:

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

A program p is constructed as follows:

$$
\begin{array}{lll}
t & ::= & a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\
g & ::= & p(t_1, \ldots, t_k) \mid X = t \\
c & ::= & p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\
p & ::= & c_1 . \ldots . c_m ? g
\end{array}
$$

- A term t either is an atom, a variable, an anonymous variable or a constructor application.
- A goal g either is a literal, i.e., a predicate call, or a unification.
- A clause c consists of a head $p(X_1, \ldots, X_k)$ with predicate name and list of formal parameters together with a body, i.e., a sequence of goals.
- A program consists of a sequence of clauses together with a single goal as query.