The code for a last call \( l \equiv (e'_0, e'_1, \ldots, e'_{n-1}) \) inside a function \( f \) with \( k \) arguments must

1. allocate the arguments \( e_i \) and evaluate \( e'_i \) 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`:

\[
\text{code;} \ (r \ \rho \ \mathsf{sd}) = \begin{aligned}
\text{code;} \ e_{n-1} & \ (r \ \mathsf{sd}) \\
\text{code;} \ e_{n-2} & \ (r \ \mathsf{sd} + 1) \\
& \ldots
\text{code;} \ e_0 & \ (r \ \mathsf{sd} + m - 1)
\end{aligned}
\]
\[
\text{code;} \ e'_0 \ (r \ \mathsf{sd} + m) \quad \text{// Evaluation of the function}
\]
\[
\text{move} \ r \ (m + 1) \quad \text{// Deallocating of \( r \) cells}
\]
\[
\text{apply}
\]

where \( r = \mathsf{sd} + k \) is the number of stack cells to deallocate.

Example

\( V \)-code for the body of the function

\[
r = \text{fun \( x, y \to \) match \( x \) with } [] \to y | h :: t \to r l (h, y)
\]

with CBN semantics:

\[
\begin{align*}
0 & \text{ targ } 2 & 1 & \text{jump } B & 4 & \text{pushglob } 0 \\
0 & \text{ pushloc } 0 & & & & 3 & \text{eval} \\
1 & \text{ eval } & 2 & A & \text{pushloc } 1 & 5 & \text{move } 4 \ 3 \\
1 & \text{ tlist } A & 3 & \text{pushloc } 4 & \text{apply} & \text{slide } 2 \\
0 & \text{ pushloc } 1 & 4 & \text{cons} & \text{pushloc } 1 & 1 & \text{slide } 2 \\
1 & \text{ eval } & 3 & \text{pushloc } 2 & \text{pushloc } 1 & 1 & \text{slide } 2 \\
\end{align*}
\]

Since the old stack frame is kept, return 2 will only be reached by the direct
jump at the end of the \( [] \) Alternative.

\[
\begin{align*}
\text{return } 2 & \to (4, 1) \\
& \to (5, 1) \\
& \to (6, 1)
\end{align*}
\]
Example

V-code for the body of the function

\[
\text{\texttt{r = fun x y \rightarrow match x with [] \rightarrow y | h :: t \rightarrow r t (h :: y)}}
\]

with CBN semantics:

\[
\begin{array}{cccc}
0 & \text{targ 2} & 1 & \text{jump B} & 4 & \text{pushglob 0} \\
0 & \text{pushloc 0} & 5 & \text{eval} \\
1 & \text{eval} & 2 & \text{A: pushloc 1} & 5 & \text{move 4 3} \\
1 & \text{tlist A} & 3 & \text{pushloc 4} & \text{apply} \\
0 & \text{pushloc 1} & 4 & \text{cons} & \text{slide 2} \\
1 & \text{eval} & 3 & \text{pushloc 1} & 1 & \text{B: return 2}
\end{array}
\]

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

214

26 Exceptions

Example

\[
\text{let rec gcd = fun x y \rightarrow}
\]

if \(x \leq 0 \lor y \leq 0\) then raise 0

else if \(x = y\) then \(x\)

else if \(y < x\) then gcd \((x - y)\) \(y\)

else gcd \(x\) \((y - x)\)

in try gcd 0 5

with \(z \rightarrow z\)

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

216

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

26 Exceptions

Example

\[
\text{let rec gcd = fun x y \rightarrow}
\]

if \(x \leq 0 \lor y \leq 0\) then raise 0

else if \(x = y\) then \(x\)

else if \(y < x\) then gcd \((x - y)\) \(y\)

else gcd \(x\) \((y - x)\)

in try gcd 0 5

with \(z \rightarrow z\)

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

216
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 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

26 Exceptions

Example

```plaintext
let rec gcd = fun x y ->
  if x <= 0 || y <= 0 then raise 0
  else if x = y then x
  else if y < x then gcd (x - y) y
  else gcd x (y - x)

in try gcd 0 5
with z -> z
```

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

216

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 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:

\[ e \equiv \text{try } e_1 \text{ with } x \rightarrow e_2 \]

we generate:

```
\text{code}_v \ e \ \rho \ \text{sd} = \quad \text{try} (A)
\text{code}_v \ e_1 \ \rho \ (\text{sd} + 4)
\text{restore} \ B \\
A: \quad \text{code}_v \ e_1 \ \rho' \ (\text{sd} + 1)
\text{slide} 1
B: \ldots
```

where \( \rho' = \rho \oplus \{ x \mapsto (L, \text{sd} + 1) \} \).
For an expression of the following form:

\[ e \equiv \text{try } e_1 \text{ with } x \rightarrow e_2 \]

we generate:

\[
\begin{align*}
\text{code}_\mathcal{V} e \rho \mathcal{S} & = \quad \text{try } A \\
\text{code}_\mathcal{V} e_1 \rho (\mathcal{S} + 4) & = \quad \text{restore } B \\
A : \quad \text{code}_\mathcal{V} e_2 \rho' (\mathcal{S} + 1) & = \quad \text{slide } 1 \\
B : \quad \ldots & = \\
\end{align*}
\]

where \( \rho' = \rho \oplus \{x \mapsto (\mathcal{L}, \mathcal{S} + 1)\} \).

Now we have all provisions to raise exceptions.

For these, we do:

- We give up the current computational context;
- We restore the context of the closest surrounding \text{try} expression;
- We hand over the exception value to the exception handler.

Thus, we translate:

\[
\begin{align*}
\text{code}_\mathcal{V} (\text{raise } e) \rho \mathcal{S} & = \quad \text{code}_\mathcal{V} e \rho \mathcal{S} \\
\text{raise} & = \\
\end{align*}
\]
Example

The code for gcd is given by:

```
0    alloc      2    B:  rewrite 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
```

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

\[ 2 \rightarrow (\ell, 2) \]
\[ \eta \rightarrow (\ell, 1) \]
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 e2;
- 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.

Caveat

Exceptions only make sense in CBV languages !

Why??

The Translation of Logic Languages
27 The Language Prolog

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

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

---

Example:

\[
\begin{align*}
\text{bigger}(X, Y) & \leftarrow X = \text{elephant}, Y = \text{horse} \\
\text{bigger}(X, Y) & \leftarrow X = \text{horse}, Y = \text{donkey} \\
\text{bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{dog} \\
\text{bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{monkey} \\
\text{is_bigger}(X, Y) & \leftarrow \text{bigger}(X, Y) \\
\text{is_bigger}(X, Y) & \leftarrow \text{bigger}(X, Z), \text{is_bigger}(Z, Y) \\
? \quad \text{is_bigger}('\text{elephant}, '\text{dog}')
\end{align*}
\]

---

A More Realistic Example:

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

---

Example:

\[
\begin{align*}
\text{bigger}(X, Y) & \leftarrow X = \text{elephant}, Y = \text{horse} \\
\text{bigger}(X, Y) & \leftarrow X = \text{horse}, Y = \text{donkey} \\
\text{bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{dog} \\
\text{bigger}(X, Y) & \leftarrow X = \text{donkey}, Y = \text{monkey} \\
\text{is_bigger}(X, Y) & \leftarrow \text{bigger}(X, Y) \\
\text{is_bigger}(X, Y) & \leftarrow \text{bigger}(X, Z), \text{is_bigger}(Z, Y) \\
? \quad \text{is_bigger}('\text{elephant}, '\text{dog}')
\end{align*}
\]
A More Realistic Example:

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

Remark:

\[
\begin{align*}
[] & \quad \text{the atom empty list} \\
[H; Z] & \quad \text{binary constructor application} \\
[a, b, Z] & \quad \text{shortcut for: } [a \mid b \mid Z] \\
\end{align*}
\]

A program \( p \) is constructed as follows:

\[
\begin{align*}
t & \quad ::= \; \text{a variable, an anonymous variable or a}& \\
& \quad \text{constructor application.} \\
g & \quad ::= \; \text{a literal, i.e., a predicate call, or a}& \\
& \quad \text{unification.} \\
\text{c} & \quad ::= \; \text{a head } p(X_1, \ldots, X_n) & \\
& \quad \text{with predicate name and list of }& \\
& \quad \text{formal parameters together with a } & \\
& \quad \text{body, i.e., a sequence of } & \\
& \quad \text{goals.} & \\
p & \quad ::= \; \text{a sequence of clauses } & \\
& \quad \text{together with a single goal as } & \\
& \quad \text{query.} & 
\end{align*}
\]

A More Realistic Example:

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

Remark:

\[
\begin{align*}
[] & \quad \text{the atom empty list} \\
[H; Z] & \quad \text{binary constructor application} \\
[a, b, Z] & \quad \text{shortcut for: } [a \mid b \mid Z] \\
\end{align*}
\]
A program $p$ is constructed as follows:

$$
t ::= a | X | \ldots | f(t_1, \ldots, t_e) \\
g ::= p(t_1, \ldots, t_k) \mid X = t \\
c ::= p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\
p ::= c_1, \ldots, c_m \
g
$$

- 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.