19  let-rec-Expressions

Consider the expression \( e = \text{let rec } y_1 = e_1 \text{ and } \ldots \text{ and } y_n = e_n \text{ in } e_0. \)
The translation of \( e \) must deliver an instruction sequence that
- allocates local variables \( y_1, \ldots, y_n; \)
- in the case of
  - CBV: evaluates \( e_1, \ldots, e_n \) and binds the \( y_i \) to their values;
  - CBN: constructs closures for the \( e_1, \ldots, e_n \) and binds the \( y_i \) to them;
- evaluates the expression \( e_0 \) and returns its value.

Caveat

In a letrec-expression, the definitions can use variables that will be allocated only later! Dummy-values are put onto the stack before processing the definition.

Example

Consider the expression
\[
e = \text{let rec } f = \text{fun } x \rightarrow \text{if } y \leq 1 \text{ then } x \text{ else } f(x+y)/(y-1) \text{ in } f
\]
for \( \rho = \emptyset \) and \( sd = 0 \). We obtain (for CBV):

\[
\begin{array}{cccc}
0 & \text{alloc} 1 & 0 & \text{A: targ 2} \\
1 & \text{pushloc} 0 & 0 & \text{...} \\
2 & \text{mkvec} 1 & 1 & \text{return 2} \\
2 & \text{mkfuncval} A & 2 & \text{rewrite 1} \\
2 & \text{jump} B & 1 & \text{mark C} \\
\end{array}
\]

For CBN, we obtain:
\[
\begin{align*}
\text{codev } e & \rho \text{ sd } = \quad \text{alloc } n \\
\text{codev } e_1 & \rho' (\text{sd} + n) \\
\text{rewrite } n \\
\ldots \\
\text{codev } e_n & \rho' (\text{sd} + n) \\
\text{rewrite } 1 \\
\text{codev } e_0 & \rho' (\text{sd} + n) \\
\text{slide } n & \quad // \text{deallocates local variables}
\end{align*}
\]
where \( \rho' = \rho \uplus \{ y_i \rightarrow (L, \text{sd} + i) \mid i = 1, \ldots, n \}. \)

In the case of CBV, we also use \text{codev} for the expressions \( e_1, \ldots, e_n. \)

Caveat

Recursive definitions of basic values are undefined with CBV!!!
The instruction **alloc n** reserves \( n \) cells on the stack and initialises them with \( n \) dummy nodes:

\[
\text{for } (i=1; i<n; i++)
S[SP+i] = \text{new } C, 1, -1;
\]
\[
SP = SP + n;
\]

Example

Consider the expression

\[
\begin{align*}
e &\equiv \text{let } r = \text{fun } f x y \rightarrow \text{if } y \leq 1 \text{ then } x \text{ else } f(x+y)(y-1) \text{ in } f(1)
\end{align*}
\]

for \( c = 0 \) and \( sd = 0 \). We obtain (for CBN):

<table>
<thead>
<tr>
<th>0</th>
<th>alloc 1</th>
<th>0: 'A'</th>
<th>2: 'B'</th>
<th>4: loadc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pushloc 1</td>
<td>5: mkbasic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mkfunval A</td>
<td>1: return 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>jump B</td>
<td>6: apply</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For CBN, we obtain:

\[
\begin{align*}
code_v \ e \ r \ s d &= \text{alloc } n & \text{// allocates local variables}
\end{align*}
\]

\[
\begin{align*}
code_c \ c_1 \ r' (s d + n)
\end{align*}
\]

\[
\begin{align*}
\text{rewrte } n
\text{slide } n &\text{// deallocates local variables}
\end{align*}
\]

where \( \rho' = \rho \oplus \{ y_i \mapsto (L, s d + i) \mid i = 1, \ldots, n \} \).

In the case of CBV, we use code_v for the expressions \( c_1, \ldots, c_n \).

Caveat

Recursive definitions of basic values are undefined with CBV!!!
Example

Consider the expression

\[ e = \text{let rec } f = \text{fun } x \ y \rightarrow \text{if } y \leq 1 \text{ then } x \text{ else } f(x + y)/(y - 1) \text{ in } f 1 \]

for \( p = 0 \) and \( sd = 0 \). We obtain (for CBV):

<table>
<thead>
<tr>
<th>0</th>
<th>alloc</th>
<th>0</th>
<th>A:</th>
<th>target 2</th>
<th>4</th>
<th>load r 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pushloc 0</td>
<td>0</td>
<td>...</td>
<td>5</td>
<td>mkbasic</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mkvec 1</td>
<td>1</td>
<td>return 2</td>
<td>5</td>
<td>pushloc 4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>mkfunval A</td>
<td>2</td>
<td>B:</td>
<td>rewrite 1</td>
<td>6</td>
<td>apply</td>
</tr>
<tr>
<td>2</td>
<td>jump B</td>
<td>1</td>
<td>mark C</td>
<td>2</td>
<td>C:</td>
<td>slide 1</td>
</tr>
</tbody>
</table>

20 Closures and their Evaluation

- Closures are needed in the implementation of CBN for let-, let-rec expressions as well as for actual parameters of functions.
- Before the value of a variable is accessed (with CBN), this value must be available.
- Otherwise, a stack frame must be created to determine this value.
- This task is performed by the instruction \( \text{eval} \).

The instruction \( \text{rewrite } n \) overwrites the contents of the heap cell pointed to by the reference at \( H[SP - n] \):

- The reference \( H[SP - n] \) remains unchanged!
- Only its contents is changed!

\( \text{eval} \) can be decomposed into small actions:

\[
\text{eval} = \text{if } (H[SP]) \equiv (C, \ldots) \{ \\
\quad \text{mark0}; \quad \text{// allocation of the stack frame} \\
\quad \text{pushloc 3}; \quad \text{// copying of the reference} \\
\quad \text{apply0}; \quad \text{// corresponds to apply} \\
\}\]

- A closure can be understood as a parameterless function. Thus, there is no need for an ap-component.
- Evaluation of the closure means evaluation of an application of this function to 0 arguments.
- In contrast to \( \text{mark A} \), \( \text{mark0} \) dumps the current PC.
- The difference between \( \text{apply} \) and \( \text{apply0} \) is that no argument vector is put on the stack.
S[SP+1] = GP;
S[SP+2] = FP;
S[SP+3] = PC;
FP = SP = SP + 3;

We thus obtain for the instruction `eval:`
The construction of a closure for an expression \( e \) consists of:

- Packing the bindings for the free variables into a vector;
- Creation of a C-object, which contains a reference to this vector and to the code for the evaluation of \( e \):

\[
\text{codeC} \ e \ \rho \ \text{sd} = \begin{cases} \text{getvar } z_0 \ \rho \ \text{sd} \\ \text{getvar } z_1 \ (\text{sd} + 1) \\ \vdots \\ \text{getvar } z_{g-1} \ (\text{sd} + g - 1) \\ \mathbf{mkvec} \ g \\ \mathbf{mkclos} \ A \\ \text{jump} \ B \\ A : \ \text{codeV} \ e \ \rho' \ 0 \\ \text{update} \\ B : \ \ldots 
\end{cases}
\]

where \( \{z_0, \ldots, z_{g-1}\} = \text{free}(e) \) and \( \rho' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g - 1\} \).

Example

Consider \( e = a \times a \) with \( \rho = \{a \mapsto (L, 0)\} \) and \( \text{sd} = 1 \). We obtain:

1. pushloc 1 0 A: pushglob 0 2 getbasic
2. mkvec 1 1 eval 2 mul
2. mkclos A 1 getbasic 1 mkbasic
2. jump B 1 pushglob 0 1 update
2. eval 2 B: ...

Example

Consider \( e = a \times a \) with \( \rho = \{a \mapsto (L, 0)\} \) and \( \text{sd} = 1 \). We obtain:

1. pushloc 1 0 A: pushglob 0 2 getbasic
2. mkvec 1 1 eval 2 mul
2. mkclos A 1 getbasic 1 mkbasic
2. jump B 1 pushglob 0 1 update
2. eval 2 B: ...
Example

Consider $e = a * a$ with $\rho = \{ a \mapsto (L, 0) \}$ and $sd = 1$. We obtain:

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Value 1</th>
<th>Value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pushloc 1</td>
<td>A</td>
<td>pushglob 0</td>
</tr>
<tr>
<td>2</td>
<td>mkvec 1</td>
<td>1</td>
<td>getbasic</td>
</tr>
<tr>
<td>2</td>
<td>mkclose A</td>
<td>1</td>
<td>getbasic</td>
</tr>
<tr>
<td>2</td>
<td>jump B</td>
<td>1</td>
<td>pushglob 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>B: ...</td>
</tr>
</tbody>
</table>

The instruction **rewrite n** overwrites the contents of the heap cell pointed to by the reference at $\text{SP} - n$:

- The reference $\text{SP} - n$ remains unchanged!
- Only its contents is changed!

21 Optimizations I: Global Variables

Observation:

- Functional programs construct many F- and C-objects.
- This requires the inclusion of (the bindings of) all global variables.
  Recall, e.g., the construction of a closure for an expression $e$...
21 Optimizations I: Global Variables

Observation:

- Functional programs construct many F- and C-objects.
- This requires the inclusion of (the bindings of) all global variables.
  Recall, e.g., the construction of a closure for an expression \( e \) ...

\[
\text{code}_C \, e \, \rho \, sd = \begin{align*}
\text{getvar} & \ z_0 \, \rho \, sd \\
\text{getvar} & \ z_1 \, \rho \, (sd + 1) \\
\ldots \\
\text{getvar} & \ z_{g-1} \, \rho \, (sd + g - 1) \\
\text{mkvec} & \ g \\
\text{mkclos} & \ A \\
\text{jump} & \ B \\
A : & \text{code}_V \, e' \, 0 \\
B : & \ldots
\end{align*}
\]

where \( \{z_0, \ldots, z_{g-1}\} = \text{free}(e) \) and \( \rho' = \{ z_i \mapsto (G, i) \mid i = 0, \ldots, g - 1 \} \).

Idea:

- **Reuse** Global Vectors, i.e. share Global Vectors!
- Profitable in the translation of let-expressions or function applications: Build one Global Vector for the union of the free-variable sets of all let-definitions resp. all arguments.
- Allocate (references to) global vectors with multiple uses in the stack frame like local variables!
- Support the access to the current \( \text{GP} \), e.g., by an instruction \( \text{copyglob} \):

\[
\text{GP} \quad \text{V} \\
\text{copyglob} \quad \text{GP} \quad \text{V}
\]

\( \text{SP}++; \)  
\( S[\text{SP}] = \text{GP}; \)
• The optimization will cause Global Vectors to contain more components than just references to the free the variables that occur in one expression ...

**Disadvantage:** Superfluous components in Global Vectors prevent the deallocation of already useless heap objects: Space Leaks :-(

**Potential Remedy:** Deletion of references from the global vector at the end of their life times.

---

### 22 Optimizations II: Closures

In some cases, the construction of closures can be avoided, namely for

- Basic values,
- Variables,
- Functions.

---

**Basic Values:**

The construction of a closure for the value is at least as expensive as the construction of the B-object itself!

Therefore:

```
code_c b o sd = code_v b o sd = load b
               mkbasic
```

This replaces:

```
mkvec 0        jump B                     mkbasic B: ...
mkclose A      A: load b                   update
```

---

**Variables:**

Variables are either bound to values or to C-objects. Constructing another closure is therefore superfluous. Therefore:

```
code_c x o sd = getvar x o sd
```

This replaces:

```
getvar x o sd  mkdles A               A: pushglob 0                    update
mkvec 1        jump B                   eval B: ...
```
Example

Consider \( e \equiv \text{let } b = 7 \text{ in } a \).

code \( e \bowtie 0 \) produces:

\[
\begin{array}{cccc}
0 & \text{alloc 2} & 3 & \text{rewrite 2} & 3 & \text{mkbasic} & 2 & \text{pushloc 1} \\
2 & \text{pushloc 0} & 2 & \text{loadc 7} & 3 & \text{rewrite 1} & 3 & \text{eval} \\
3 & \text{slide 2} & & & & & & \\
\end{array}
\]

The execution of this instruction sequence should deliver the basic value 7...
let rec \( \text{\textbf{c}} = \overline{a} \) and \( \overline{a} = \text{\textbf{b}} \) in \( \overline{a} \)

Apparently, this optimization was not quite correct. :

The Problem:

Binding of variable \( y \) to variable \( x \) before \( x \)'s dummy node is replaced!!

The Solution:

- **cyclic definitions**: reject sequences of definitions like
  - let rec \( a = b \) and \( \ldots b = a \) in \( \ldots \)
- **acyclic definitions**: order the definitions \( y = x \) such that the dummy node for the right side of \( x \) is already overwritten.

Functions:

Functions are values, which are not evaluated further. Instead of generating code that constructs a closure for an F-object, we generate code that constructs the F-object directly.

Therefore:

\[
\text{code}_c \ (\text{fun } x_0 \ldots x_{n-1} \rightarrow e) \ r \ s d = \ \text{code}_y \ (\text{fun } x_0 \ldots x_{n-1} \rightarrow e) \ r \ s d
\]
Remarks:

- The code schemata as defined so far produce Spaghetti code.
- Reason: Code for function bodies and closures placed directly behind the instructions mkfunval resp. mkclos with a jump over this code.
- Alternative: Place this code somewhere else, e.g. following the halt-instruction:

  Advantage: Elimination of the direct jumps following mkfunval and mkclos.

  Disadvantage: The code schemata are more complex as they would have to accumulate the code pieces in a Code-Dump.

Solution:

Disentangle the Spaghetti code in a subsequent optimization phase. :)

23 The Translation of a Program Expression

Execution of a program \( e \) starts with

\[
\begin{align*}
PC &= 0 \\
SP &= FP = GP = -1
\end{align*}
\]

The expression \( e \) must not contain free variables.

The value of \( e \) should be determined and then a \texttt{halt} instruction should be executed.

\[
\begin{align*}
\text{code } e &= \text{code}_y\ e \ 0 \\
&\quad \text{halt}
\end{align*}
\]