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:

\[
\text{code}_V \ b \ \rho \ \text{sd} = \text{code}_V \ b \ \rho \ \text{sd} = \text{loadc} \ b
\]

\[
\text{mkbasic}
\]

This replaces:

\[
\text{mkvec} \ 0 \quad \text{jump} \ B \quad \text{mkbasic} \quad B: \quad ...
\]

\[
\text{mkclos} \ A \quad A: \quad \text{loadc} \ b \quad \text{update}
\]
Variables:

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

\[
\text{code}_C \ x \ \rho \ \text{sd} \ = \ \text{getvar} \ x \ \rho \ \text{sd}
\]

This replaces:

\[
\text{getvar} \ x \ \rho \ \text{sd} \quad \text{mkclos} \ A \quad A: \quad \text{pushglob} \ 0 \quad \text{update}
\]
\[
\text{mkvec} \ 1 \quad \text{jump} \ B \quad \text{eval} \quad B: \quad ...
\]
Example:

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

code\( \lambda \)\( e \) \( \emptyset \) 0 produces:

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

The execution of this instruction sequence should deliver the basic value 7 ...
0 alloc 2 3 rewrite 2 3 mkbasic 2 pushloc 1
2 pushloc 0 2 loadc 7 3 rewrite 1 3 eval 3 slide 2
0 alloc 2 3 rewrite 2 3 mkbasic 2 pushloc 1
2 pushloc 0 2 loadc 7 3 rewrite 1 3 eval
3 slide 2

pushloc 0

```
<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th></th>
<th>alloc 2</th>
<th></th>
<th>rewrite 2</th>
<th></th>
<th>mkbasic</th>
<th></th>
<th>pushloc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>pushloc 0</td>
<td>2</td>
<td>loadc 7</td>
<td>3</td>
<td>rewrite 1</td>
<td>3</td>
<td>eval</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td>slide 2</td>
</tr>
</tbody>
</table>

**Diagram:**

```
<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>−1</th>
<th>−1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>−1</td>
<td>−1</td>
<td></td>
</tr>
</tbody>
</table>
```

**loadc 7**

```
<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>−1</th>
<th>−1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>−1</td>
<td>−1</td>
<td></td>
</tr>
</tbody>
</table>
```
0 alloc 2 3 rewrite 2 3 mkbasic 2 pushloc 1
2 pushloc 0 2 loadc 7 3 rewrite 1 3 eval 3 slide 2

7

C -1 -1
C -1 -1

mkbasic
<table>
<thead>
<tr>
<th></th>
<th>Alloc 2</th>
<th></th>
<th>Rewrite 2</th>
<th></th>
<th>Mkbasic</th>
<th></th>
<th>Pushloc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Pushloc 0</td>
<td>2</td>
<td>Loadc 7</td>
<td></td>
<td>Rewrite 1</td>
<td>3</td>
<td>Eval</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Slide 2</td>
</tr>
</tbody>
</table>

The diagram shows a stack of boxes labeled B and C, with values -1 and 7, indicating a computational process. The arrows indicate the flow of data or instructions.
0 alloc 2
2 pushloc 0
3 rewrite 2
2 loadc 7
3 rewrite 1
3 eval
3 slide 2

pushloc 1

B 7
C −1 −1
0 alloc 2  
2 pushloc 0  
2 loadc 7  
3 rewrite 2  
3 loadc 7  
3 rewrite 1  
2 pushloc 1  
3 eval  
3 slide 2

\[
\begin{align*}
B & \quad 7 \\
C & \quad -1 \quad -1
\end{align*}
\]
0 alloc 2 3 rewrite 2 3 mkbasic 2 pushloc 1
2 pushloc 0 2 loadc 7 3 rewrite 1 3 eval
3 slide 2

Segmentation Fault !!
Apparently, this optimization was not quite correct  :-(

The Problem:

Binding of variable $y$ to variable $x$ before $x$’s dummy node is replaced!!

$\Rightarrow$

The Solution:

- **cyclic definitions**: reject sequences of definitions like
  
  ```
  let a = b; \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:

\[
\text{code}_C (\text{fun } x_0 \ldots x_{k-1} \rightarrow e) \rho \text{ sd} = \text{code}_V (\text{fun } x_0 \ldots x_{k-1} \rightarrow e) \rho \text{ sd}
\]
The Translation of a Program Expression

Execution of a program $e$ starts with

$$PC = 0 \quad SP = FP = GP = -1$$

The expression $e$ must not contain free variables.

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

$$\text{code } e = \text{code}_V e \emptyset 0$$

\texttt{halt}
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 :-)}
Example: \( \text{let } a = 17 \text{ in let } f = \text{fun } b \rightarrow a + b \text{ in } f 42 \)
24 Structured Data

In the following, we extend our functional programming language by some datatypes.

24.1 Tuples

Constructors: $(\ldots, \ldots, \ldots)$, \(k\)-ary with \(k \geq 0\);

Destructors: \(#j\) for \(j \in \mathbb{N}_0\) (Projections)

We extend the syntax of expressions correspondingly:

\[
e ::= \ldots \mid (e_0, \ldots, e_{k-1}) \mid #j\ e \\
\mid \text{let } (x_0, \ldots, x_{k-1}) = e_1 \text{ in } e_0
\]
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 \texttt{mkvec}.

For returning components we use an indexed access into the tuple.

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

\[
\text{code}_V (\# j e) \rho \text{ sd} = \text{code}_V e \rho \text{ sd} \\
\text{get} j \\
\text{eval}
\]

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

197
if (S[SP] == (V,g,v))
  S[SP] = v[j];
else Error "Vector expected!";
Inversion: Accessing all components of a tuple simultaneously:

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

This is translated as follows:

\[
\text{code}_V \ e \ \rho \ \text{sd} \ = \ \text{code}_V \ e_1 \ \rho \ \text{sd} \\
\quad \text{getvec} \ k \\
\quad \text{code}_V \ e_0 \ \rho' \ (\text{sd} + k) \\
\quad \text{slide} \ k
\]

where \( \rho' = \rho \oplus \{y_i \mapsto (L, \text{sd} + i + 1) \mid i = 0, \ldots, k - 1\} \).

The instruction \textbf{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!";
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`:

\[
app = \text{fun } l \ y \rightarrow \text{match } l \text{ with } \\
\quad | \ [ ] \rightarrow y \\
\quad | \ h :: t \rightarrow h :: (\text{app } t \ y)
\]
accordingly, we extend the syntax of expressions:

\[
e ::= \ldots \mid [] \mid (e_1 :: e_2) \mid (\text{match } e_0 \text{ with } [] \rightarrow e_1 \mid h :: t \rightarrow e_2)
\]

Additionally, we need new heap objects:

- \(\text{Nil}\) - empty list
- \(s[0]\), \(s[1]\) - non-empty list
24.3 Building Lists

The new instructions \texttt{nil} and \texttt{cons} are introduced for building list nodes.

We translate for CBN:

\[
\begin{align*}
\text{code}_V \ [\ ] \ \rho \ sd &= \text{nil} \\
\text{code}_V \ (e_1 :: e_2) \ \rho \ sd &= \text{code}_C \ e_1 \ \rho \ sd \\
&\quad \text{code}_C \ e_2 \ \rho \ (sd + 1) \ \\
&\quad \text{cons}
\end{align*}
\]

Note:

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

203
SP++; S[SP] = new (L,Nil);
S[SP-1] = new (L, Cons, S[SP-1], S[SP]);
SP - -;
24.4 Pattern Matching

Consider the expression \( e \equiv \text{match } e_0 \text{ with } [] \rightarrow e_1 \mid h :: t \rightarrow e_2. \)

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):

\[
\text{code}_V e \rho \ sd = \text{code}_V e_0 \rho \ sd \\
\text{tlist} \ A \\
\text{code}_V e_1 \rho \ sd \\
\text{jump} \ B \\
A : \ \text{code}_V e_2 \rho' (sd + 2) \\
\text{slide} \ 2 \\
B : \ ... \\
\]

where \( \rho' = \rho \oplus \{h \mapsto (L, sd + 1), t \mapsto (L, sd + 2)\} \).

The new instruction \text{tlist} \ A does the necessary checks and (in the case of Cons) allocates two new local variables:
\[
\begin{align*}
h &= S[SP]; \\
&\text{if } (H[h] \neq (L,\ldots)) \\
&\quad \text{Error "no list!";} \\
&\text{if } (H[h] = (_,\text{Nil})) \ SP- -; \\
&\ldots
\end{align*}
\]
... else {
    S[SP+1] = S[SP] \rightarrow s[1];
    S[SP] = S[SP] \rightarrow s[0];
    SP++; PC = A;
}

Example: The (disentangled) body of the function \( \text{app} \) with
\[ \text{app} \mapsto (G, 0) : \]

<table>
<thead>
<tr>
<th>0</th>
<th>targ 2</th>
<th>3</th>
<th>pushglob 0</th>
<th>0</th>
<th>C:</th>
<th>mark D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>pushloc 0</td>
<td>4</td>
<td>pushloc 2</td>
<td>3</td>
<td>pushglob 2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>eval</td>
<td>5</td>
<td>pushloc 6</td>
<td>4</td>
<td>pushglob 1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>tlist A</td>
<td>6</td>
<td>mkvec 3</td>
<td>5</td>
<td>pushglob 0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>pushloc 1</td>
<td>4</td>
<td>mkclos C</td>
<td>6</td>
<td>eval</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>eval</td>
<td>4</td>
<td>cons</td>
<td>6</td>
<td>apply</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>jump B</td>
<td>3</td>
<td>slide 2</td>
<td>1</td>
<td>D:</td>
<td>update</td>
</tr>
<tr>
<td>2</td>
<td>A:</td>
<td>pushloc 1</td>
<td>1</td>
<td>B:</td>
<td>return 2</td>
<td></td>
</tr>
</tbody>
</table>

Note:

Datatypes with more than two constructors need a generalization of the \text{tlist} instruction, corresponding to a \text{switch}-instruction

:-)
24.5 Closures of Tuples and Lists

The general schema for \( \text{code}_C \) can be optimized for tuples and lists:

\[
\begin{align*}
\text{code}_C (e_0, \ldots, e_{k-1}) \, \rho \, sd &= \text{code}_V (e_0, \ldots, e_{k-1}) \, \rho \, sd \\
&= \text{code}_C e_0 \, \rho \, sd \\
&= \text{code}_C e_1 \, \rho \, (sd + 1) \\
&\vdots \\
&= \text{code}_C e_{k-1} \, \rho \, (sd + k - 1) \\
&\text{mkvec} \, k
\end{align*}
\]

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