22 Optimiizations II: Closures

In some cases, the construction of closures can be avoided, namely for
- Basic values,
- Variables,
- Functions.

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 at the end of their life time.

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

Therefore:
\[
\text{codeg } b \ p \ sd = \text{codev } b \ p \ sd = \text{loadc } b \ \text{mkbasic}
\]

This replaces:
\[
\text{mkvec } 0 \quad \text{jump } B \quad \text{mkbasic } B \quad \text{...}
\]
\[
\text{mkclose } A \quad \text{A: loadc } b \quad \text{update}
\]
Variables:

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

\[
\text{code: } \text{x \rho \delta} \Rightarrow \text{getvar } x \rho \delta
\]

This replaces:

\[
\text{getvar } x \rho \delta \quad \text{mkclos } A \quad \text{pushglob } 0 \quad \text{update}
\]

\[
\text{mkvec } 1 \quad \text{jump } B \quad \text{eval } \quad \text{B: } \ldots
\]

Example:

\[
e \equiv \text{let rec } a = b \text{ and } b = 7 \text{ in } a.
\]

\[
\text{code: } e 0 0
\]

produces:

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

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 load 7  3 rewrite 1  3 eval  3 slide 2

Segmentation Fault !!

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

\[ \text{codcx} \ x \ \rho \ \mathsf{sd} \quad \Rightarrow \quad \text{getvar} \ x \ \rho \ \mathsf{sd} \]

This replaces:

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

Example:
\[ c = \text{let rec } a = b \text{ and } b = 7 \text{ in } a. \quad \text{codcx} \ c \ 0 \ 0 \]
produces:
0 alloc 2  3 rewrite 2  3 mkbasic  2 pushloc 1
2 pushloc 0  2 load 7  3 rewrite 1  3 eval  3 slide 2
The execution of this instruction sequence should deliver the basic value 7.

Segmentation Fault!!
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:

\[
\text{code}_C (\text{fun } x_0 \ldots x_{k-1} \rightarrow e) \; \rho \; \text{sd} = \; \text{code}_Y (\text{fun } x_0 \ldots x_{k-1} \rightarrow e) \; \rho \; \text{sd}
\]
23 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 $\text{halt}$ instruction should be executed.

\[
\text{code } e = \text{codey } e \oplus 0 \\
\text{halt}
\]

Remarks:

- The code schemata as defined so far produce $\text{Spaghetti code}$.
- Reason: Code for function bodies and closures placed directly behind the instructions $\text{mkfunval}$ resp. $\text{mkclos}$ with a jump over this code.
- Alternative: Place this code somewhere else, e.g. following the $\text{halt}$-instruction.
  - Advantage: Elimination of the direct jumps following $\text{mkfunval}$ and $\text{mkclos}$.
  - 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 $\text{:-)}$

Example:

\[
\begin{array}{cccccccc}
0 & \text{loadc} & 17 & 2 & \text{mark} & B & 3 & \text{B} & \text{slide} & 2 & 1 & \text{pushloc} & 1 \\
1 & \text{mbkbasic} & 5 & \text{loadc} & 42 & 1 & \text{halt} & 2 & \text{eval} \\
1 & \text{pushloc} & 0 & 6 & \text{mbkbasic} & 0 & A: & \text{tang} & 1 & 2 & \text{getbasic} \\
2 & \text{mkvec} & 1 & 6 & \text{pushloc} & 4 & 0 & \text{pushglob} & 0 & 2 & \text{add} \\
2 & \text{mkfunval} & 1 & 7 & \text{eval} & 1 & \text{eval} & 1 & \text{mbkbasic} \\
7 & \text{apply} & 1 & \text{getbasic} & 1 & \text{return} & 1 & \text{eval} & 1 & \text{getbasic} & 1 & \text{return} & 1 \\
\end{array}
\]
24 Structured Data

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

24.1 Tuples

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

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

We extend the syntax of expressions correspondingly:

$$ e ::= \ldots \mid (e_0, \ldots, e_{n-1}) \mid \text{let } (x_0, \ldots, x_{n-1}) = e_1 \text{ in } e_0 $$

In order to construct a tuple, we collect a sequence of references on the stack. Then we construct a vector of these references in the heap using `mkvec`.

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

$$ \text{codey } (e_0, \ldots, e_{n-1}) \rho \text{ sd} = \text{codey } e_0 \rho \text{ sd} $$
$$ \text{codey } e_1 \rho \text{ (sd + 1)} \ldots $$
$$ \text{codey } e_{n-1} \rho \text{ (sd + k - 1)} \text{ mkvec } k $$

$$ \text{codey } (\#j \ e) \rho \text{ sd} = \text{codey } e \rho \text{ sd} $$
$$ \text{get } j $$
$$ \text{eval } $$

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

\[196\]

\[197\]

Inversion: Accessing all components of a tuple simultaneously:

$$ e \equiv \text{let } (y_0, \ldots, y_{n-1}) = e_1 \text{ in } e_0 $$

This is translated as follows:

\[198\]

if (SISP == (Vg, v))
SISP = vj;
else Error "Vector expected!";

$\rho = \rho \oplus \{y_i \mapsto (L, sd + i + 1) \mid i = 0, \ldots, k - 1\}$. The instruction \texttt{getvec k} pushes the components of a vector of length $k$ onto the stack.

\[199\]
24.2 Lists

Lists are constructed by the constructors:
- `[]` “Nil”, the empty list;
- `"a"` “Cons”, right-associative, takes an element and a list.

Access to list components is possible by match-expressions ...

Example: The append function `app`:

\[
\text{app} = \text{fun } l y \rightarrow \text{match } l \text{ with } \\
\quad [] \rightarrow y \\
\quad h::t \rightarrow h::(\text{app } t \ y)
\]

```
201
```

accordingly, we extend the syntax of expressions:

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

Additionally, we need new heap objects:

- `L Nil` empty list
- `L Cons` non-empty list

```
202
```

24.3 Building Lists

The new instructions `nil` and `cons` are introduced for building list nodes.

We translate for CBN:

\[
\begin{align*}
\text{code}_{\times} \ [] & \rho \ s d = \text{nil} \\
\text{code}_{\times} \ (e_1::e_2) & \rho \ s d = \text{code}_{\times} \ e_1 \rho \ s d \\
& \quad \quad \text{code}_{\times} \ e_2 \rho \ (s d + 1) \\
& \quad \quad \text{cons}
\end{align*}
\]

Note:
- With CBN: Closures are constructed for the arguments of “;”;
- With CBV: Arguments of “;” are evaluated :-)

```
203
```

SP++; S[SP] = new (L,Nil);

```
204
```
24.3 Building Lists

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

\[
\text{code}_\text{C} \ \ [] \ \ \rho \ \sigma d = \text{nil} \\
\text{code}_\text{C} \ \ {e_1} :: {e_2} \ \rho \ \sigma d = \text{code}_\text{C} \ e_1 \ \rho \ \sigma d \ \downarrow \text{code}_\text{C} \ e_2 \ \rho \ ((\sigma d + 1)) \ \text{cons}
\]

Note:
- With CBN: Closures are constructed for the arguments of "$\cdot$";
- With CBV: Arguments of "$\cdot$" are evaluated $\Rightarrow$

24.4 Pattern Matching

Consider the expression $e = \text{match } e_0 \text{ with } [] \rightarrow e_1 \ | \ h :: t \rightarrow e_2$.

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):

\[
\begin{align*}
\text{code} e & \rho \text{ sd} = \text{code} e_0 & \rho \text{ sd} \\
\text{tlist A} & \\
\text{code} e_1 & \rho \text{ sd} \\
\text{jump B} & \\
A : & \text{code} e_2 & \rho' \text{ (sd + 2)} \\
& \text{slide 2} & \\
B : & \ldots & \\
\end{align*}
\]

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:

---

In consequence, we obtain (for CBN as for CBV):

\[
\begin{align*}
\text{code} e & \rho \text{ sd} = \text{code} e_0 & \rho \text{ sd} \\
\text{tlist A} & \\
\text{code} e_1 & \rho \text{ sd} \\
\text{jump B} & \\
A : & \text{code} e_2 & \rho' \text{ (sd + 2)} \\
& \text{slide 2} & \\
B : & \ldots & \\
\end{align*}
\]

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.
24.5 Closures of Tuples and Lists

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

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

\[
\text{code}_t [] \; \rho \; sd = \text{code}_t [] \; \rho \; sd = \text{nil} \\
\text{code}_t (e_1 : e_2) \; \rho \; sd = \text{code}_t (e_1 : e_2) \; \rho \; sd = \text{code}_t e_1 \; \rho \; sd \\
\text{code}_t e_2 \; \rho \; (sd + 1) \\
\text{cons}
\]

25 Last Calls

A function application is called \texttt{last call} in an expression \( e \) if this application could deliver the value for \( e \).

A last call usually is the outermost application of a defining expression.

A function definition is called \texttt{tail recursive} if all recursive calls are last calls.

Examples:

\( r \; t \; (h :: y) \) is a last call in\( \texttt{match} x \; \texttt{with} \; [] \rightarrow y \; | \; h \; :: t \rightarrow r \; (h :: y) \)

\( f \; (x - 1) \) is not a last call in\( \texttt{if} \; x \leq 1 \; \texttt{then} \; 1 \; \texttt{else} \; x \; \times \; f \; (x - 1) \)

Observation: Last calls in a function body need no new stack frame!