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 $e$ ...

In fact, the instruction \texttt{update} is the combination of the two actions:

\begin{verbatim}
popenv
rewrite 1
\end{verbatim}

It overwrites the closure with the computed value.

\texttt{codeC} $e \ 0 \ sd $ = \begin{align*}
\text{getvar} \ z_0 \ p \ sd \\
\text{getvar} \ z_1 \ p \ (sd + 1) \\
\vdots \\
\text{getvar} \ z_{g-1} \ p \ (sd + g - 1) \\
kvec \ g \\
\mkclos \ A \\
\text{jump} \ B \\
A : \ \text{codeV} \ e \ p' \ 0 \\
\text{update} \\
B : \ \vdots
\end{align*}

where $\{z_0, \ldots, z_{g-1}\} = \text{free} (e)$ and $p' = \{z_i \mapsto (G, i) \mid i = 0, \ldots, g - 1\}$. 
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 GP, e.g., by an instruction `copyglob`:

```
SP++;  
S[SP] = 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  \( \Longrightarrow \) Space Leaks

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

\[
\begin{align*}
\forall \psi \in \Gamma & \quad \text{if } \psi \text{ is a basic value, a variable, or a function} \\
\forall \psi \in \Gamma & \quad \text{if } \psi \text{ is a basic value, a variable, or a function}
\end{align*}
\]

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: \[ \text{code}_\varepsilon b \rho \sigma d = \text{code}_\varepsilon b \rho \sigma d = \text{loadc}_b \text{mkbasic} \]

This replaces:

\[ \begin{align*}
\text{mkvec } 0 & \quad \text{jump } B \\
\text{mkclos } A & \quad \text{loadc } b \quad \text{update} \\
\end{align*} \]

Example

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

\[ \text{code}_\varepsilon e \emptyset 0 \text{ produces:} \]

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

The execution of this instruction sequence should deliver the basic value 7 ...

Variables

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

\[ \text{code}_\varepsilon x \rho \sigma d = \text{getvar}_x \rho \sigma d \]

This replaces:

\[ \begin{align*}
\text{getvar } x \rho \sigma d & \quad \text{mkclos } A \\
\text{mkvec } 1 & \quad \text{jump } B \\
A & \quad \text{pushglob } 0 \\
A & \quad \text{update} \\
& \quad \text{eval} \\
B & \quad \ldots
\end{align*} \]
Example
Consider $e \equiv \text{let rec } a = 0 \text{ end } b = 7 \text{ in } a$. code $e \neq 0$ produces:

```
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
```

The execution of this instruction sequence should deliver the basic value 7 ...
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 ... b = a in ...

acyclic definitions: order the definitions y = x such that the dummy node for the
right side of x is already overwritten.

23 The Translation of a Program Expression

Execution of a program e starts with
\( FC = 0 \quad SP = FP = GP = -1 \)

The expression e must not contain free variables.
The value of e should be determined and then a \( \text{halt} \) instruction should be executed.

\[
\text{code } e = \text{code}_{\gamma} \ e \ 0 \ 0 \\
\text{halt}
\]

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:

\[
\text{code}_{\epsilon} (\text{fun } x_0 \ldots x_{k-1} \rightarrow e) \ \rho \ \sigma \ = \ \text{code}_{\gamma} (\text{fun } x_0 \ldots x_{k-1} \rightarrow e) \ \rho \ \sigma
\]

Remarks

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

let \( a = 17 \)

\[
\text{let } f = \text{fun } b \to a + b \text{ in } 42
\]

Disentanglement of the jumps produces:

\[
\begin{array}{cccccc}
0 & \text{loads} & 17 & 2 & \text{mark B} & 3 & \text{B: slide 2} & 1 & \text{pushloc 1} \\
1 & \text{mkbasic} & 5 & \text{load 42} & 1 & \text{halt} & 2 & \text{eval} \\
1 & \text{pushloc 0} & 6 & \text{mkbasic} & 0 & \text{AC} & 1 & \text{getbasic} \\
2 & \text{mkvec 1} & 6 & \text{pushloc 4} & 0 & \text{pushbb 0} & 2 & \text{add} \\
2 & \text{mkfunval} & 7 & \text{eval} & 1 & \text{eval} & 1 & \text{mkbasic} \\
7 & \text{apply} & 1 & \text{getbasic} & 1 & \text{return} & 1 & \\
\end{array}
\]

24 Structured Data

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

24.1 Tuples

Constructors: \((\ldots, a, \ldots, a_n)\) with \(n \geq 0\);

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

We extend the syntax of expressions correspondingly:

\[
e ::= \ldots | (a_0, \ldots, a_{n-1}) | \#j e
\]

\[
\text{let } (x_0, \ldots, x_{n-1}) = e_1 \text{ in } e_0
\]

- 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 \text{mkvec}

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

\[
\text{codev} (a_0, \ldots, a_{n-1}) \rho \text{ sd} = \text{codev} a_0 \rho \text{ sd}
\]

\[
\text{codev} a_i \rho (\text{sd} + 1)
\]

\[
\ldots
\]

\[
\text{codev} a_{n-1} \rho (\text{sd} + k - 1)
\]

\[
\text{mkvec } k
\]

\[
\text{codev} (\#j e) \rho \text{ sd} = \text{codev} e \rho \text{ sd}
\]

\[
\text{get } j
\]

In the case of \text{CBV}, we directly compute the values of the \(e_i\).
**Inversion:** Accessing all components of a tuple simultaneously:

\[ \epsilon \equiv \text{let } (y_0, \ldots, y_{k-1}) = e_1 \text{ in } e_2 \]

This is translated as follows:

\[
\begin{align*}
\text{code}_V \ e \ p \ s d & = \text{code}_V e_1 \ p \ s d \quad \text{getvec } k \\
& \quad \text{code}_V e_0 \ p' \ (s \ d + k) \\
& \quad \text{slide } k
\end{align*}
\]

where \( p' = p \oplus \{ y_i \mapsto (L, s \ d + i + 1) \mid i = 0, \ldots, k-1 \} \).

The instruction \text{getvec } k \text{ pushes the components of a vector of length } k \text{ onto the stack:}

---

if \( [\text{SP}] == (V k v) \) |
    \text{SP}--; 
for(i=0; i<k; i++) |
    \text{SP}++; [\text{SP}] = v[i];
} |
else Error "Vector expected!";
24.2 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:

\[
\text{app} = \text{fun } l y \rightarrow \text{match } l \text{ with }
\]

\[
| \[] \rightarrow y \\
| h :: t \rightarrow h :: (\text{app } t \ y)
\]

accordingly, we extend the syntax of expressions:

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

Additionally, we need new heap objects:

- L Nil: empty list
- L Cons: non-empty list

SP++: S[SP] = new (L, Nil)
24.4 Pattern Matching

Consider the expression $e \equiv \mathtt{match} \ e_0 \ 	ext{with} \ [\_ \rightarrow e_1] \ h :: t \rightarrow \ e_2$.

Evaluation of $e$ requires:
- evaluation of $e_0$;
- check, whether resulting value $\nu$ is an L-object;
- if $\nu$ is the empty list, evaluation of $e_1$ ...
- otherwise storing the two references of $\nu$ on the stack and evaluation of $e_2$. This corresponds to binding $h$ and $t$ to the two components of $\nu$.

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

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

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

The new instruction $\text{tlist A}$ does the necessary checks and (in the case of Cons) allocates two new local variables:
Example

The (disentangled) body of the function app with app → (G, 0):

| 0 | tag 2 | 3 | pushglob 0 |
| 0 | pushloc 0 | 4 | pushloc 2 |
| 1 | eval | 5 | pushloc 6 |
| 1 | tlist A | 6 | mkvec 3 |
| 0 | pushloc 1 | 4 | mkvec C |
| 1 | eval | 4 | cons |
| 1 | jump B | 3 | slide 2 |
| 2 | A: pushloc 1 | 1 | B: return 2 |

Example

The (disentangled) body of the function app with app → (G, 0):

| 0 | tag 2 | 3 | pushglob 0 |
| 0 | pushloc 0 | 4 | pushloc 2 |
| 1 | eval | 5 | pushloc 6 |
| 1 | tlist A | 6 | mkvec 3 |
| 0 | pushloc 1 | 4 | mkvec C |
| 1 | eval | 4 | cons |
| 1 | jump B | 3 | slide 2 |
| 2 | A: pushloc 1 | 1 | B: return 2 |

Remark

Datatypes with more than two constructors need a generalization of the tlist instruction, corresponding to a switch-instruction.
24.5 Closures of Tuples and Lists

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

$$\text{code}_C (n, \ldots, n_{-1}) \rho \text{sd} = \text{code}_V (n, \ldots, n_{-1}) \rho \text{sd} = \frac{\text{code}_C \ n \ rho \ \text{sd}}{\text{code}_C \ n_1 \ (\text{sd} + 1)}$$

$$\ldots$$

$$\text{code}_C n_{-1} \ n \ (\text{sd} + k - 1)$$

$\text{mkvec} \ k$

$$\text{code}_C [] \ n \ \text{sd} = \text{code}_V [] \ n \ \text{sd} = \text{nil}$$

$$\text{code}_C (n_1 :: n_2) \ n \ \text{sd} = \frac{\text{code}_V (n_1 :: n_2) \ n \ \text{sd}}{\text{code}_C \ n_1 \ rho \ \text{sd}}$$

$$\text{code}_C \ n_2 \ n \ (\text{sd} + 1)$$

$\text{cons}$