6 Implementing functional languages

In this section we will describe how to compile programs in a functional language and execute them on a small abstract machine defined by a transition system. The language will be λ-calculus extended by natural numbers and addition. Other extensions would be straightforward, so we prefer to keep the language small.

As an introduction we define a semantic function for λ-calculus. Let **Lambda** be the set of all expressions according to the abstract grammar

$$e ::= x \mid \lambda x.e \mid e_1\ e_2$$

where $x \in \mathbf{Var}$ is an identifier. We will use applicative reduction order and the meaning of an expression will a λ-expression on normal form if the reduction produces one.

$$\mathcal{S}_{ap} \in \mathbf{Lambda} \hookrightarrow \mathbf{Lambda}$$

The definition will use an auxiliary function, apply.

$$\begin{aligned}
&\mathcal{S}_{ap}[\![x]\!] \triangleq x \\
&\mathcal{S}_{ap}[\![\lambda x.e]\!] \triangleq \lambda x.\mathcal{S}_{ap}[\![e]\!] \\
&\mathcal{S}_{ap}[\![e_1\ e_2]\!] \triangleq \text{apply } (\mathcal{S}_{ap}[\![e_1]\!])\ (\mathcal{S}_{ap}[\![e_2]\!]) \\
&\text{apply } (\lambda x.e_1)\ e_2 \triangleq \mathcal{S}_{ap}[\![e_1[x \mapsto e_2]]\!] \\
&\text{apply } e_1\ e_2 \triangleq e_1\ e_2, \qquad \text{if } e_1 \text{ is not an abstraction}
\end{aligned}$$

The actual computation is performed in the fourth line with $e_1[x \mapsto e_2]$. This definition is highly recursive and is not appropriate for a low level implementation.

We will provide a low level implementation in two steps. First we will compile the expression into a machine language for a computer with four registers that can contain different kinds of lists.

In order to be able to compute numerical values without representing them by lambda expressions we extend the grammar.

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid n \mid e_1 + e_2$$

where n, as usual is a numeral denoting a natural number.

The set of instructions, **Instr**, is defined by the grammar

$$i ::= \text{NUM } n \mid \text{LOAD } x \mid \text{ADD} \mid \text{AP} \mid \text{FUN}(x, c)$$

A code sequence, **Code**, is a list of instructions

$$c ::= \epsilon \mid i : c$$

Our computer model will have a stack for making computations and NUM n will push the value of n on the stack.

Similarly LOAD x will push the value of x onto the stack. The value will be retrieved from a variable environment which is just a list of pairs. Each pair will have a variable name and the corresponding value.

The ADD instruction will replace the two topmost values on the stack with their sum.

The FUN instruction will push a representation of an abstraction and the current environment, a *closure*, on the stack to be used by the AP instruction.

The AP instruction will take a closure and a value from the stack and "apply" the closure to the value. The details will be specified by transition rules for the computer.

Next we define the compiling function, $\mathcal{C} \in \mathbf{Expr} \rightarrow \mathbf{Code}$.

$$\mathcal{C}[\![n]\!] \triangleq \text{NUM } n$$
$$\mathcal{C}[\![x]\!] \triangleq \text{LOAD } x$$
$$\mathcal{C}[\![e_1 + e_2]\!] \triangleq \mathcal{C}[\![e_1]\!] : \mathcal{C}[\![e_2]\!] : \text{ADD}$$
$$\mathcal{C}[\![e_1\ e_2]\!] \triangleq \mathcal{C}[\![e_1]\!] : \mathcal{C}[\![e_2]\!] : \text{AP}$$
$$\mathcal{C}[\![\lambda x.e]\!] \triangleq \text{FUN}(x, \mathcal{C}[\![e]\!])$$

We are going to consider one example,

(twice double) 3

where

double $\triangleq \lambda x.x + x$
twice $\triangleq \lambda f.\lambda x.f(f\ x)$

Expr does not permit us to name expressions, so formally we are considering

$(\lambda f.\lambda x.f(f\ x))\ (\lambda x.x + x)\ 3$

and expecting it to deliver

$(\lambda f.\lambda x.f(f\ x))\ (\lambda x.x + x)\ 3 =$
$(\lambda x.(\lambda x.x + x)((\lambda x.x + x)\ x))\ 3 =$
$(\lambda x.x + x)((\lambda x.x + x)\ 3) = (\lambda x.x + x)6 = 6 + 6 = 12$

We start to compile double and twice.

$\mathcal{C}[\![\text{double}]\!] =$ `FUN(x,`$\mathcal{C}$`[[x+x]])= FUN(x, LOAD x:LOAD x:ADD)`
$\mathcal{C}[\![\text{twice}]\!] =$ `FUN(f, FUN(x,`$\mathcal{C}$`[[f (f x)]]))= FUN(f, FUN(x ,LOAD f:LOAD f:LOAD x:AP:AP)`

The full expression compiles to

$\mathcal{C}[\![(\text{twice double})\ 3]\!] = \mathcal{C}[\![(\text{twice}]\!] : \mathcal{C}[\![\text{double}]\!]$`:AP:NUM 3:AP` $=$

```
FUN(f, FUN(x, LOAD f : LOAD f : LOAD x : AP : AP) :
FUN(x, LOAD x : LOAD x : ADD) :
AP : NUM 3 : AP
```

The computer has four registers, S (the stack), E (the environment), C (the code register), and D (the dump). Each of the register will hold a list of values. In a hardware representation each register may contain an address to the location of the list in the main memory.

The stack is used to hold intermediate values during the computation. The environment register holds information on previous substitutions. The code register contains the instructions to be executed. The dump is used for preserving code and environments while reducing local redexes. The contents of the registers have the following types.

$$\textbf{Stack} \triangleq \textbf{Value}^*$$
$$\textbf{Env} \triangleq (\textbf{Var} \times \textbf{Value})^*$$
$$\textbf{Code} \triangleq \textbf{Instr}^*$$
$$\textbf{Dump} \triangleq (\textbf{Env} \times \textbf{Code})^*$$

There are to kinds of values on the stack, natural numbers and closures.

$$\textbf{Value} \triangleq \mathrm{N} \cup (\textbf{Var} \times \textbf{Code} \times \textbf{Env})$$

A configuration of the computer is defined by the contents of the four registers, i.e. a configuration is an element in $\textbf{Stack} \times \textbf{Env} \times \textbf{Code} \times \textbf{Dump}$ or is a terminal configuration defined by a single value. The possible execution steps are defined by the relation $\rhd$ on configurations.

$(v : \epsilon, env, \epsilon, \epsilon) \rhd v$
$(s, env, \texttt{NUM}\ n : c, d) \rhd (\mathcal{N}[\![n]\!] : s, env, c, d)$
$(s, env, \texttt{LOAD}\ x : c, d) \rhd (\text{lookup}\ env\, x : s, env, c, d), \qquad$ if x occurs in env
$(n_1 : n_2 : s, env, \texttt{ADD} : c, d) \rhd (n_1 + n_2 : s, env, c, d)$
$(s, env, \texttt{FUN}\ (x, c') : c, d) \rhd ((x, c', env) : s, env, c, d)$
$(v : (x, c', env') : s, env, \texttt{AP} : c, d) \rhd (s, (x, v) : env', c', (env, c) : d)$
$(s, env, \epsilon, (env', c') : d) \rhd (s, env', c', d)$

lookup $env\ x$ will find the value of x in the environment env',

$$\mathit{lookup}\ ((x', v') : env)\ x \triangleq \text{if } x = x' \text{ then } v' \text{ else } \mathit{lookup}\ env\ x$$

If the final result of an execution is an abstraction, the machine will not reduce the body and it will return a closure.

Executing the code for our example expression we get in each execution step:

$S = \epsilon \quad E = \epsilon \quad C = \texttt{FUN(f,}$ twc $\texttt{):}$ dbl $\texttt{:AP :NUM 3:AP} \quad D = \epsilon$

where dbl $= \texttt{FUN (x, LOAD x:LOAD x:ADD)}$
and twc $= \texttt{FUN(x, LOAD f:LOAD f:LOAD x:AP :AP)}$

$S = (\texttt{f,}$ twc $, \epsilon) \quad E = \epsilon \quad C = \texttt{FUN(x, LOAD x:LOAD x:ADD):AP :NUM 3:AP} \quad D = \epsilon$

$S = \texttt{(x, LOAD x:LOAD x:ADD ,}\epsilon\texttt{), (f,}$ twc $\texttt{,}\epsilon) \quad E = \epsilon \quad C = \texttt{AP :NUM 3:AP} \quad D = \epsilon$

$S = \epsilon \quad E = env1 \quad C =$ twc $\quad D = (\epsilon, \texttt{NUM 3:AP})$

where $env1 = \texttt{(f, (x,LOAD x:LOAD x:ADD ,}\epsilon\texttt{))}$

$S = \texttt{(x,LOAD f:LOAD f:LOAD x:AP :AP ,}env1) \quad E = env1 \quad C = \epsilon \quad D = (\epsilon, \texttt{NUM 3:AP})$

$S = $ (x, LOAD f:LOAD f:LOAD x:AP :AP ,$env1$) $E = \epsilon$ $C =$ NUM 3:AP $D = \epsilon$

$S =$ 3:(x,LOAD f:LOAD f:LOAD x:AP :AP ,$env1$) $E = \epsilon$ $C =$ AP $D = \epsilon$

$S = \epsilon$ $E = env2$ $C =$ LOAD f:LOAD f:LOAD x:AP :AP $D = (\epsilon, \epsilon)$

where $env2 =$ (x, 3),(f,(x,LOAD x:LOAD x:ADD ,ϵ))

$S =$ (x, LOAD x:LOAD x:ADD ,ϵ) $E = env2$ $C =$ LOAD f:LOAD x:AP :AP $D = (\epsilon, \epsilon)$

$S =$ (x, LOAD x:LOAD x:ADD,ϵ),$clo1$ $E = env2$ $C =$ LOAD x:AP :AP $D = (\epsilon, \epsilon)$

where $clo1 =$ (x,LOAD x:LOAD x:ADD ,ϵ) $S =$ 3:(x,LOAD x:LOAD x:ADD ,ϵ),$clo1$ $E = env2$ $C =$ AP :AP $D = (\epsilon, \epsilon)$

$S =$ (x,LOAD x:LOAD x:ADD ,ϵ) $E =$ (x: 3) $C =$ LOAD x:LOAD x:ADD $D = (env2, \text{AP}), (\epsilon, \epsilon)$

$S =$ 3,(x,LOAD x:LOAD x:ADD ,ϵ) $E = (x, 3)$ $C =$ LOAD x:ADD $D = (env2, \text{AP}), (\epsilon, \epsilon)$

$S =$ 3: 3:(x,LOAD x:LOAD x:ADD ,ϵ) $E =$ (x, 3) $C =$ ADD $D = (env2, \text{AP}), (\epsilon, \epsilon)$

$S =$ 6:(x,LOAD x:LOAD x:ADD ,ϵ) $E =$ (x, 3) $C = \epsilon$ $D = (env2, \text{AP}), (\epsilon, \epsilon)$

$S =$ 6:(x,LOAD x:LOAD x:ADD ,ϵ) $E = env2$ $C =$ AP $D = (\epsilon, \epsilon)$

$S = \epsilon$ $E =$ (x, 6) $C =$ LOAD x:LOAD x:ADD $D = (env2, \epsilon), (\epsilon, \epsilon)$

$S =$ 6 $E =$ (x, 6) $C =$ LOAD x:ADD $D = (env2, \epsilon), (\epsilon, \epsilon)$

$S =$ 6: 6 $E =$ (x, 6) $C = ADD$ $D = (env2, \epsilon), (\epsilon, \epsilon)$

$S =$ 12 $E =$ (x, 6) $C = \epsilon$ $D = (env2, \epsilon), (\epsilon, \epsilon)$

$S =$ 12 $E = env2$ $C = \epsilon$ $D = (\epsilon, \epsilon)$

$S =$ 12 $E = \epsilon$ $C = \epsilon$ $D = \epsilon$