Abstractions

Every model of computation is also often supplemented with (one or more) *abstraction* mechanisms. Abstractions are “recipes” to create modular program components — they are the building blocks of almost all computer science methodologies for programming. Programming can be viewed as the systematic composition and instantiation of various abstractions, whether on data or with control abstractions.

We have seen how expressions with variables — namely formulas — can yield different values/answers when the variables are bound to different values/answers. Formulas are useful when they can be used multiple times.

Similarly different types of data can be formed using “parametric” type constructions such as cartesian product, or lists, or user-defined parametric types such as trees. The specific components of these constructions can be filled in for the identified type parameters — we can form, e.g., integer lists, boolean lists, lists of integer lists, etc. by instantiating the list type construction with different type instances.

When forming an *abstraction*, we identify certain variable parts as “*parameters*”. These parameters are instantiated during invocation of the abstraction to yield different instances. For example, we can take an expression such as $x^2 - 2x + y$ containing the variable x, which we designate as a parameter, and then create a function in that variable x with the “body” being the given expression. If this function is invoked with the parameter $x = 3$, we get the expression $3 + y$, whereas if we invoke it with parameter $x = -4$, we get the expression $24 + y$. Note that we need not identify all variables in an expression as parameters; for example the variable y is not a parameter in this function we have created. In a static scoping (lexical scoping) discipline, this variable y is “global” or “free” with respect to this function abstraction. It takes its value from a table that is “outside” the scope of the function body. In contrast, in the function abstraction which we have created (on parameter x), x is treated as “bound” within the function.

The usefulness of such expression abstractions is that (i) we can invoke the abstract form with different argument values, which is useful when doing the same computation in different contexts, or evaluating the same expressions for different argument values for the parameter; (ii) we evaluate all instantiations (instances) of the expression in the same way — so we an compile the expression to the same code, and we can formulate a uniform mechanism for forming abstracts.

Similarly, in the case of forming types, we can use a uniform construction method for creating different instances, e.g., lists of different element types such as integer lists, boolean lists, etc. We can identify a common set of functions which are associated with all kinds of lists, irrespective of the type of the elements in the list — functions such as `length` of a list, or finding the first element and remaining part of a non-empty list. Moreover, we can define useful functions such as applying a given function to all members of a list (the function `map`), or combining the elements of a list using a binary operator with a default value (the functions `fold_left` or `fold_right`).

From a software engineering viewpoint, such abstractions, whether on expression or on types, are extremely useful, since we can reuse the code, minimising the length of the programs that we write, as well as analysing the programs for correctness and efficiency once — rather than having to do so for each instance. The minimises both the effort in programming as well as the potential for introducing bugs in the the software.

In the context of our toy programming language, an abstraction expression is a "function" with certain variables being parameters, and the expression being the body of the function. In imperative languages such as Algol and Pascal (and C, C++, and Java), programs are built out of "commands" such as assignments, conditional commands, iterations, etc. The abstract forms are called "procedures". In Pascal, C and C++, there are forms of procedures which return arguments — these are somewhat confusingly also called "functions", but "value-returning procedures" is perhaps better terminology.

Principle of Abstraction

There is an underlying principle that governs how abstractions of various kinds can be introduced into a language — as well as suggesting how they can be implemented.

The *Principle of Abstraction* states that one can form abstracts for *any* meaningful syntactic category (e.g., expressions, commands, types, ...). The abstract is a conceptual template that can be used and reused whenever necessary. This is called "invocation" — and the *invocation of an abstract* of syntactic category *C* is an instance of category *C*. For example, the invocation of a function is an expression; the invocation of a procedure is a command; the invocation of a type abstraction is a type, etc.

In their purest forms, abstracts need not have any parameters, but it is most common to designate certain identifiers within an element of a syntactic category as "parameters", and when invoking the abstract, we provide "*arguments*" (belonging to the same syntactic category as the identified parameter).

Most languages use their definition mechanism to allow the programmer to name abstracts. For example, we can name an expression abstract (i.e., a function), and can invoke the abstract by using its name (and in most cases, providing it with the necessary arguments). In our toy language, we would create a syntactic category for expression abstracts (functions), and extend the abstract syntax of expressions to include invocations of expression abstracts (more commonly known as "*function call*").

Note that according to the Principle of Abstraction, an abstract on syntactic category *C* need not be an element of syntactic category *C* — it is only the invocation of the abstract (with arguments) that must be an element of syntactic category *C*. For example, in C, a function is not an expression. Moreover, a function cannot be anonymous; when defining a function, we <u>must</u> give it a name.

In a hypothetical language with expressions, one could create *parameter-less* abstracts — and present them as two mutually-recursively defined syntactic categories:

$$e \in Exp \ ::= \ \ldots \mid \textbf{invoke}\ a$$
$$a \in Abs \ ::= \ \lceil e \rceil$$

The expression with the expression abstract $a = \lceil e \rceil$ is not evaluated until it is explicitly *invoked;* and when invoked, the abstracted expression *e* is evaluated its value returned as a result.

It is more common to have abstracts with parameters — we use the notation $\lambda(x_1, \ldots, x_n) . \lceil e \rceil$, where $(x_1, \ldots, x_n)$ are (*binding-occurrences*) of the identified parameters that may appear in the body of the abstract $\lceil e \rceil$. All instances of $x_1, \ldots, x_n$ in the body $\lceil e \rceil$ are *bound occurrences* of these variables in the body. The Greek letter λ is used as a *binder*.

First-class Citizens

However, there are many languages where abstracts (including parameterised abstracts) are treated as "first-class citizens". That is, the abstracts need not be given a name via a definition mechanism, and so can be invoked anonymously (i.e., without using a name). Languages such as SML, OCaml and Haskell allow us to treat expression abstracts, i.e., functions as expressions — that is they are "first-class", in that they can be used wherever expression (of the appropriate type) can be used — as arguments to other functions, as results returned by function calls, in definitions, as elements of data structures such as lists and trees, etc. And they do not *have* to be given a name in all such contexts (being and staying anonymous is as much a right of first-class citizens as anything else).

When abstracts are first-class citizens, the syntactic category C and that of abstracts on C are combined. We will consider a simple form of abstraction where an abstract can have only one identified parameter (we will see that restriction causes no loss of generality):

$$e \in Exp \ ::= \ \ldots \mid \lambda x . e \ \mid \ (e_1 \ e_2)$$

where we drop the delimiters $\lceil \_ \rceil$ around the body e of the abstract, and instead of writing the verbose **invoke** e_1 **with** e_2 to denote the invocation of expression e_1 with argument expression e_2, we write the notationally lighter $(e_1 \ e_2)$. [There are certain advantages of writing invocation this way, in preference to $e_1(e_2)$ — the more common form used in mathematics.

A possible data type definition in OCaml to represent an abstract syntax for expressions and their abstracts is

```
        type exp = … | Abs of string * exp   |  App of exp * exp ;;
```

Typing Rules

In the functional paradigm, the type given to functions corresponds to *function spaces*. Let $A, B \subseteq \mathscr{U}$ be subsets of a universal set $\mathscr{U}$. We define the function space

$$[A \rightharpoonup B] = \{f \in \mathscr{U} \rightharpoonup \mathscr{U} \mid \text{if } a \in A \text{ and } (a,b) \in \text{graph}(f), \text{ then } b \in B\}$$

i.e, all those functions which, given *any* argument $a \in A$, whenever they return a result $b = f(a)$, then it is guaranteed that $b \in B$. It is instructive to think of the function space construction as a *constraint* on what set an answer (value) is guaranteed to belong. (We will see later, when we talk about subtypes, that this formulation is a more handy one than a purely set theoretic formulation (with domains and ranges specified).

We need expand our class of types with a new type construction, namely the function type:

$$\tau, \tau_1, \tau_2 \in Typ \ ::= \ \ldots \mid \tau_1 \to \tau_2$$

$$\textbf{(AbsT)} \ \frac{\Gamma[\underline{x} : \tau_1] \vdash \underline{e} : \tau_2}{\Gamma \vdash \underline{\lambda x . e} : \tau_1 \to \tau_2} \qquad\qquad \textbf{(AppT)} \ \frac{\Gamma \vdash \underline{e_1} : \tau_1 \to \tau_2 \qquad\qquad \Gamma \vdash \underline{e_2} : \tau_1}{\Gamma \vdash \underline{(e_1 \ e_2)} : \tau_2}$$

The typing rule (**AppT**) is fairly easy to understand: If under the same type assumptions Γ, "operator" expression e_1 has a function type $\tau_1 \to \tau_2$, and "argument" expression e_2 has type τ_1, then the function invocation $(e_1\ e_2)$ — of function e_1 on argument e_2 — has type τ_2. The typing rule (**AbsT**) describes how functions are formed: If we augment the typing assumption Γ with the assumption of formal parameter x having some type τ_1, and under this augmented type assumption, we find that the body of the function e has type τ_2, then the abstract (function) $\lambda x\,.\,e$ has a function type $\tau_1 \to \tau_2$.

The denotational definition is extended with the following case:

$$eval[\![\underline{\lambda x\,.\,e}]\!]\,\rho = f \in [A \rightharpoonup B]$$

where $f(a) = eval[\![\,\underline{e}\,]\!](\rho[x \mapsto a])$ for each $a \in A$

$$eval[\![\,\underline{(e_1\ e_2)}\,]\!]\,\rho = f(a)$$

where $f = eval[\![\underline{e_1}]\!]\rho$ and $a = eval[\![\underline{e_2}]\!]\rho$

The operational specification is somewhat more intricate because of the possible presence of "free variables" in an expression $\lambda x\,.\,e$ — in a static scoping discipline, these variables take their meaning from the lexical context in which the expression is written. For example, the expression $\lambda x\,.\,x + y$ is intended to mean a function that adds the value of y to whatever argument value we give for parameter x. But what value of y should we use? In the lexical (static/textual) scoping discipline, if this function was defined in a context where y was bound to 7, then irrespective of which context we invoked the function, it would add 7 to the given argument.

So the following naïve inference rules are *not* correct.

(**CalcAbsWrong**) $$\frac{}{\gamma \vdash \underline{\lambda x\,.\,e} \Longrightarrow \underline{\lambda x\,.\,e}}$$

(**CalcAppWrong**) $$\frac{\gamma \vdash \underline{e_1} \Longrightarrow \underline{\lambda x\,.\,e_1'} \qquad \gamma \vdash \underline{e_2} \Longrightarrow \underline{a_2} \qquad \gamma[\underline{x} \mapsto \underline{a_2}] \vdash \underline{e_1'} \Longrightarrow \underline{a}}{\gamma \vdash \underline{(e_1\ e_2)} \Longrightarrow \underline{a}}$$

(There are good reasons to follow a lexical scoping discipline, since we can reason about a program without having to worry about the run-time scope in which it is executed. If there is any information from the run-time environment that needs to be used, we can have an explicit formal parameter so that those values can be passed to the function in a principled manner. However, this is *not* to say that for certain applications, dynamic binding is more convenient — e.g., in displaying text in a word processing document, we may implicitly pick up the position, width and height, size and font from the context — since it is a pain to list all of these as parameters and to pass them explicitly. But be warned that reasoning about programs in a framework with dynamic binding is always prone to error, including type errors.)

Exercise Give an example where by using these incorrect rules, we would violate the lexical scoping discipline.

Closures

To correctly implement the lexical scoping discipline, we somehow have to correctly bind all the (apparently) free variables which appear in a function body to their values/answers in the prevailing environment. Note that when we invoke a function with an actual argument, the intended operation is to *substitute the argument for the formal parameter in the body of the function*. Recall that we had introduced the "table" data structure so that variables could be quickly and efficiently looked up (in quick time) … instead of the expensive operation of substituting answers (values) for each variable.

So if there are "global" variables in the body of a function, i.e., not parameters, and not locally defined using a let-definition, we should be able to find the bindings for these global variables in the prevailing table. Now instead of performing expensive substitution operations, we can "pack in" the table with the function code, and whenever we encounter a global variable, look up its binding in the table. The data structure consisting of the function expression and the table together is called a "*closure*" (because it closes up all the apparently free variable references).

$$Clos = Exp \times Table$$

with function closures represented as $\langle\langle \lambda x . e,\ \gamma \rangle\rangle$. We add such closures which we designate by $VClos = \{\lambda x . e \ \mid\ x \in \mathscr{X}, e \in Exp\} \times Table$ into our set of canonical answers.

We are now in a position to write the correct Big-step rules for function abstractions and application:

(CalcAbs) $$\frac{}{\gamma \vdash \underline{\lambda x . e} \Longrightarrow \langle\langle \lambda x . e,\ \gamma \rangle\rangle}$$

That is, an abstraction calculates to a closure as a canonical answer, packing in the current table into the closure. Of course, in a good implementation, one would not copy a table and create an unwieldy and large data structure, but instead insert a reference to the table. Some program analysis may also help to only keep relevant bindings from the table, and cut away all the unnecessary flab.

(CalcApp) $$\frac{\gamma \vdash \underline{e_1} \Longrightarrow \langle\langle \lambda x . e',\ \gamma' \rangle\rangle \qquad \gamma \vdash \underline{e_2} \Longrightarrow \underline{a_2} \qquad \gamma'[\underline{x} \mapsto \underline{a_2}] \vdash \underline{e'} \Longrightarrow \underline{a}}{\gamma \vdash \underline{(e_1\ e_2)} \Longrightarrow \underline{a}}$$

The operational semantics rule (**CalcApp**) details how the result of a function call $(e_1\ e_2)$ is calculated: First we calculate the answer for the "operator" expression e_1, which being a function, should result in a closure. Note that this answer closure is of the form $\langle\langle \lambda x . e',\ \gamma' \rangle\rangle$ where the expression component is an abstraction of the form $\lambda x . e'$, where x is the formal parameter, and e' the body of the function. Note also that the table packed into the closure *may* be different from the table γ with respect to which we calculated the answer (for example, it may be the result for looking up the table for a named function, which was created in another environment), and hence we indicate this possibly different table as γ'. Next we evaluate the "argument" expression e_2, in the same call-time environment, namely table γ as the operator expression, and let us call the obtained answer a_2. Now (in this call-by-value discipline), we bind the argument answer a_2 to the formal parameter x, and then evaluate the body e' of the function. But this evaluation of e' is not done with respect to the call-time environment γ, but rather with the environment γ' saved in the closure — from where we will get the correct bindings in a lexical/statically scoped discipline for the global variables that appear in body e' — which is augmented by the binding of the formal parameter to the actual argument, namely $x \mapsto a_2$. The answer a obtained from this evaluation of the function body is returned as the answer of the function call.

You may have a noticed that we included (some) closures — which contain tables as a component — into our set of answers. And tables mapped variables to answers. So there is some mutual recursion in these definitions:
$Clos = Exp \times Table$ and $Table = \mathcal{X} \rightharpoonup_{fin} Ans$, where $VClos \subset Ans$.

Exercise: Write a function *unpack* from closures to expressions, that recursively unpacks a closure into an expression, by substituting for a variable appearing in the expression component of a closure the unpacking of the closure to which it is bound in the table of the closure.

Once you have done so, you may wish to try your hand at the following:

Exercise: Prove the new induction cases in the Soundness Theorem.

Exercise: Prove the new induction cases in the Completeness Theorem.

The Type Preservation theorem also continues to hold!

Exercise: Prove the new induction cases in the Type Preservation Theorem.
Caution: One has to be quite careful in this proof, in stating and using the IH.

Alternative Formulation: Big-step as closure evaluation

Recall that in our formulation of big-step natural semantics with a table, we had written the table γ to the left of a turnstile $\vdash$ to emphasise that the table did not change. But here in the (**CalcApp**) rule, we switch to a possibly different table γ' which is what prevailed when the closure was created, augment γ' with the formal-parameter-to-actual-argument binding in order to calculate the body of the function, we revert to the call-time table γ once we have obtained the answer a, we revert to the call-time table.

There is an alternative formulation of these rules where all answers are closures. tables are mappings from variables to closures, and we present all calculation as happening on closures ... where closures in $Clos$ are transformed into a subset of “canonical closures” in $VClos$.

(**CalcVar’**) $\dfrac{}{\langle\langle \underline{x}, \gamma\rangle\rangle \Longrightarrow_{clos} \gamma(x)}$

where the variable x is looked up in table γ, and we obtain a closure as the result.

(**CalcAbs’**) $\dfrac{}{\langle\langle \lambda x\,.\,e,\ \gamma\rangle\rangle \Longrightarrow_{clos} \langle\langle \lambda x\,.\,e,\ \gamma\rangle\rangle}$

is a trivial rule, and application is rewritten in terms of closure simplification as:

(CalcApp')

$$\frac{\langle\langle \underline{e_1}, \gamma\rangle\rangle \Longrightarrow_{clos} \langle\langle \lambda x . e', \gamma'\rangle\rangle \qquad \langle\langle \underline{e_2}, \gamma\rangle\rangle \Longrightarrow_{clos} vcl_2 \qquad \langle\langle \underline{e'}, \gamma' [\underline{x} \mapsto vcl_2]\rangle\rangle \Longrightarrow_{clos} vcl}{\langle\langle \underline{(e_1\ e_2)}, \gamma\rangle\rangle \Longrightarrow_{clos} vcl}$$

Now you have a choice to make about how to represent numeric constants such as the integer 3: Is it to be an answer as it is, or should we represent it as a closure of the form $\langle\langle 3, \gamma\rangle\rangle$ for any table γ (in particular, $\langle\langle 3, \varnothing\rangle\rangle$)? The former seems more space-efficient, the latter provides a simpler, uniform definition of tables: $Clos = Exp \times Table$ and $Table = \mathscr{X} \rightharpoonup_{fin} VClos$. In fact, for the slightly different semantics of lazy evaluation (call-by-name), we will have $Table = \mathscr{X} \rightharpoonup_{fin} Clos$.

Exercise: Write a function *unpack* from closures to expressions, that recursively unpacks a closure into an expression, by substituting for a variable appearing in the expression component of a closure the unpacking of the closure to which it is bound in the table of the closure.

Once you have done so, you may wish to try your hand at the following:

Exercise: Prove the new induction cases in the Soundness Theorem.

Exercise: Prove the new induction cases in the Completeness Theorem.

The Type Preservation theorem also continues to hold!

Exercise: Prove the new induction cases in the Type Preservation Theorem.
Caution: One has to be quite careful in this proof, in stating and using the IH.

Compilation and Stack Machine Execution

How should the Stack Machine be modified to deal with closure formation and function call?

Peter J. Landin in the period 1964-66 wrote two very influential papers:

- The mechanical evaluation of expressions

- The Next 700 Programming Languages

The first paper « is a contribution to the "theory" of the activity of using computers. It shows how some forms of expression used in current programming languages can be modelled in Church's λ-notation, and then describes a way of "interpreting" such expressions. This suggests a method, of analyzing the things computer users write, that applies to many different problem orientations and to different phases of the activity of using a computer. Also a technique is introduced by which the various composite information structures involved can be formally characterized in their essentials, without commitment to specific written or other representations.»

Ideas such as abstract syntax, syntactic sugar, reducing a bigger language to a core sub-language, interpreters, byte code, compilation as a homomorphic traversal of an abstract syntax tree, and execution using a stack machine ... and many other seminal ideas are discussed in this paper.

The SECD Machine is a machine for executing *op-codes* obtained from *compiling* an expression from a high level language.

Language. For simplicity, let us consider the λ-calculus, i.e., the core sublanguage

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

We can represent this language in OCaml as a data type:

```
type exp = V of string  |  Abs of string * exp  | App of exp * exp;;
```

This core language can be extended to embrace numerals, boolean constants, the unit value, pairs, injections, projections from pairs, if-then-else conditional expressions, case statements, etc.

In the sequel, we will assume that the expressions that we compile are well-typed and (at the top level) have no free variables.

Opcodes
We introduce a notion of opcodes, which will be the elementary operations of a run-time abstract machine.

```
type opcode =  LOOKUP(x)  |  APP  | MKCLOS(x, c)  |  RET;;
```

If we extend the core language, we will have to (re-)introduce opcodes for performing the operations at the machine level corresponding to the evaluation of the expression.

A compiled program is again going to be a list of opcodes. Note that the single opcode MKCLOS(x, c) has nested in it a list c of op-codes. (In practical implementations, this will be represented using a reference to a list, so that the opcode can be represented as a fixed size opcode. Also variables x will eventually be converted into memory addresses or register references).

Compilation. Expressions from our language are compiled into opcode sequences. As usual, we specify the *compiler* as a recursive function (a homomorphism, which happens to be a post-order traversal of the abstract syntax tree) as follows:

compile(x) = [`LOOKUP`(x)]
compile($\lambda x . e_1$) = [`MKCLOS`(x, *compile*(e_1) @ `[RET]`)]
compile($(e_1\ e_2)$) = *compile*(e_1) @ *compile* (e_2) @ [`APP`]

Notice that the compilation of applications is very similar to how one would compile an arithmetic expression $e_1 + e_2$ for a stack evaluator, i.e., compiling first e_1 then e_2 and then appending at the end an op-code such as `PLUS`.

More importantly, note that the compilation of λ-abstractions involves a recursive call to compile the body of the function e_1 but this is nested within the `MKCLOS` opcode. Note also that we append at the very end of the compiled body the `RET` opcode, to mark that the result of the function call must be returned to the calling context.

Exercise: Code the compile function in OCaml.

The components of the machine. The SECD machine is named for its constituent 4 components:

S — "Stack" stands for a stack of *answers,* corresponding to the sub-expressions which have already been evaluated so far.

E —"Environment" (which we will here write as a *table* γ), a mapping from variables $\mathcal{X}$ to answers *Ans.*

C — "Code" is a list of opcodes, and corresponds to the rest of the program that remains to be executed.

D — "Dump" is a stack of *(S, E, C)* triples, which represent the context to which control has to return after completion of a function call. (The λ-calculus is after all a higher-order functional language). (Again, in practice, we will not stack up such triples of large data-structures, but instead will use references to them).

Closures
At the machine level, since a syntactic expression has been compiled into an opcode sequence, we present a "machine-oriented" version of closures: as triples $vcl \in VClosI ::= \langle\langle\langle x, c, \gamma \rangle\rangle\rangle$, where x is a variable (the formal parameter), c is an opcode sequence, and γ is a table, where tables are finite-domain functions from variables to answers, with $VClosI \subseteq Ans$

We will see below that the opcode `MKCLOS`(x, c) will make this form of a "value closure" by packing in the current table (environment).

Exercise: Define a suitable data type representation of closures. Note that closures and tables will be mutually recursively defined.

Configurations of the SECD Machine

The configurations of the SECD machine are quadruples of the form

$$(S, \gamma, c, D)$$

where
- S is a stack (representable as a list) of *Ans;*
- $\gamma \in \mathcal{X} \rightharpoonup Ans$; (tables may be representable as finite domain functions or as lists)
- c is an op-code list; and
- D is a stack of *(S, E, C)* triples (again, a dump can be represented as a list of triples)

Note that in an efficient implementations, we will not actually stack up (S, E, C) triples, but only references to them.

Transition Rules of the SECD Machine.

The operation of the machine is presented as a single step transitions, based on the opcode at the head of the *C* component, There is only one rule for each opcode, and whether a transition is possible may depend on the state of some of the other component.
If the configuration does not match the expected shape, then the machine gets *stuck*.

- Look up a variable in the table.

$(S, \gamma, \texttt{LOOKUP}(x) :: c', D) \quad ==> \quad (a::S, \gamma, c' D),$ *provided x in dom(γ) and $a = \gamma(x)$.*

- Make a closure by packing in the current table, and place it on the stack.

$(S, \gamma, \texttt{MKCLOS}(x, c') :: c'', D) ==> (\langle\langle\langle x, c', \gamma' \rangle\rangle\rangle :: S, \gamma, c'', D)$

- Bind the actual argument a (at the top of the stack) to the formal parameter x, and add it to the environment (table) γ' that was packed in the operator closure $\langle\langle x, c', \gamma' \rangle\rangle$ that should be just below the top of the stack. Then execute the code c' of the operator closure, starting with an <u>empty</u> stack. Save the calling context (S, γ, c'') by pushing it onto the dump.

$(a::\langle\langle\langle x, c', \gamma' \rangle\rangle\rangle::S, \; \gamma, \; \texttt{APP} :: c'', \; D) ==> ([\,], \; \gamma' [x \mapsto a], c', \; (S, \gamma, c'') :: D)$

- Return from the function call by restoring — and popping — the calling context (S, γ, c'') from the dump, placing the answer a at the top of the restored stack S, and resuming executing the code c'' with the restored table γ. Note that we discard the context components S', c' and γ'' of the called function since they are no longer needed.

$(a::S', \; \gamma'', \; \texttt{RET} :: c', \; (S, \gamma, c'') :: D) ==> (a::S, \; \gamma, c'', \; D)$

Programming Exercise. Implement the SECD machine in OCaml — taking the (reflexive) transitive closure of the one-step transitions.

Exercise (**). Formulate a theorem that states that if a (closed well-typed) expression *e* evaluates to an answer *a*, then by compiling *e* into opcodes and running the compiled code on the SECD machine will result in a state where something analogous to *a* is on the top of the stack.

Exercise (***) Prove the theorem that you formulate. On what will you perform induction? Why? What generalisations on stack, environment, code and dump parameters will you need to make for the induction hypotheses to be applicable?