A LANGUAGE WITH CONTEXT

A crucial feature of all modern languages is the support for nested contexts in which names, whether of variables, types or subprograms, can be re-used. This gives the programmer much more flexibility when it comes to breaking the program down into modules, each of which is like a whole program in its own right. Nested contexts bring with them the concepts of local and non-local references. Pascal is the prototypical language of this kind, where all names need to be declared before they are used, and all name references can be disambiguated by the rules of static scope. The Pascal skeleton program below uses the name x four times in different contexts, referring to three different variables.

```
program scope;
  var x;
  procedure A;
    var x;
    begin
      x := 1
    end;
  procedure B;
    var x;
    procedure C;
    begin
      x := 2;
    end;
    begin
      x := 3
    end;
    begin
      x := 4;
    end.
```

We will examine the denotational semantics of static scoping by augmenting the simple language (without types) that we studied earlier. In order to do this we will need to add new domains, some new semantic algebras, and change the valuation functions accordingly.

LOCATIONS AND THE ENVIRONMENT

Different contexts need syntax to indicate the where one context stops and the next one starts. Pascal uses nested blocks to indicate these regions, surrounded by begin-end markers (like the braces in C, C++ and Java). If uses of names within different blocks are to have different meanings (i.e. refer to different objects) then the names must be attached to the block before their use. This means that names must be declared before they are used. This separation of declaration from use is characteristic of these languages, and leads to a separation of names from values as well. Previously we mapped variable names directly to values in the Store with a function Store → Nat. We will need to separate out the business of declaration from the business of assignment because the former is static, whereas the latter is dynamic. In particular, we need to be able to restore the context at the end of a block so that further assignments will be made to the correct variable.

So what is the meaning of a variable name before it is used? In the real machine this is handled by allocating different memory addresses for variables. We will abstract this idea in a primitive domain of locations. Although real locations are mapped onto numerical addresses (taken from the natural numbers, starting from 0 and increasing) we will not go that far, and leave locations as primitive objects. The domain can be defined by:

Domain Location
Operations
  first-locn: Location
  next-locn: Location → Location
  equal-locn: Location → Location → Tr
  less-locn: Location → Location → Tr

The idea here is that we can always get to the next location by using next-locn and the set is anchored at first-locn.
The meaning of names will now be expressed in a domain called the *Environment*, which is where locations are stored:

Domain $\text{Environment} = \text{Id} \to (\text{Location} + \text{Errorvalue})$, where $\text{Errorvalue} = \text{Unit}$

Operations

- $\text{emptyenv}: \text{Environment} = \lambda \lambda . \lambda . \lambda_i \in \text{Errorvalue}()$
- $\text{accessenv}: \text{Id} \to \text{Environment} \to (\text{Location} + \text{Errorvalue})$
- $\text{updateenv}: \text{Id} \to \text{Location} \to \text{Environment} \to \text{Environment}$

Values are now stored in a mapping with the variable’s location. We will still call this the store, even though it is now a different function:

Domain $\text{Store} = \text{Location} \to \text{Nat}$

Operations

- $\text{access}: \text{Location} \to \text{Store} \to \text{Nat}$
- $\text{update}: \text{Location} \to \text{Nat} \to \text{Store} \to \text{Store}$

Any expression now needs to be evaluated relative to two objects, the environment and the store. The names of variables will be looked up in the environment to retrieve the variable’s location (different for different contexts). The location can then be used to store different values at different times. The first mapping is fixed, and governed purely by the static context; the second is dynamic and can change during program execution. As we shall see, this is exactly the distinction between compile-time and run-time maintained by many compilers. The environment is rather like the symbol-table maintained by the compiler to keep track of names declared in different contexts. This can (usually) be discarded at run-time since we can only alter the store; the name become irrelevant.

**A BLOCK-STRUCTURED LANGUAGE**

The syntax for our new language will include the block as a new category. It is Pascal-like, but whereas Pascal declares its variables outside the block’s begin-end, we have put them inside, like Algol, and all the C based languages. There is also the possibility of declaring a named constant, which will also be placed in the environment. There is no if-then-else, but that is not the focus of our language.

\[
P \in \text{Program} \\
K \in \text{Block} \\
D \in \text{Declaration} \\
C \in \text{Command} \\
E \in \text{Expression} \\
B \in \text{Boolean-expr} \\
I \in \text{Identifier} \\
Ν \in \text{Numeral} \\
\]

\[
P ::= K. \\
K ::= \text{begin } D; \ C \text{ end} \\
D ::= D_1; D_2 \mid \text{const } I = N \mid \text{var } I \\
C ::= C_1; C_2 \mid I := E \mid \text{while } B \text{ do } C \mid K \\
E ::= E_1 + E_2 \mid I \mid N \\
\]
Since we now have declarations, there are two error situations to look out for. One is trying to assign to an undeclared variable, and the other is trying to use the value of an undeclared variable. Both of these are handled in the semantic algebras:

**Domain** Nat (definition as before)

**Domain** Tr (definition as before)

**Domain** Id (definition as before)

**Domain** Location (definition above)

**Domain** Expressible-value = Nat + Errorvalue,  
where Errorvalue = Unit

**Domain** Denotable-value = Location + Nat + Errorvalue,  
where Errorvalue = Unit

We have added Nat to the denotable values so that named constants can be stored in the environment. The Errorvalue will take care of undeclared names (variables or constants).

The environment will be slightly altered to be a pair consisting of the map from identifiers to locations, and the next highest location. reserve-locn will take the environment pair, and return the next-highest location from the pair, and a new environment with the next highest location increased by using next-locn.

**Domain** Environment = (Id → Denotable-value) × Location

**Operations**

- emptyenv: Location → Environment
  \[ emptyenv = \lambda l.((\lambda i.\text{inErrorvalue}()) , l) \]

- accessenv: Id → Environment → Denotable-value
  \[ \text{accessenv} = \lambda i.\lambda [\text{map}, l].\text{map}(i) \]

- updateenv: Id → Denotable-value → Environment → Environment
  \[ \text{updateenv} = \lambda i.\lambda d.\lambda [\text{map}, l].([i \mapsto d].\text{map}, l) \]

- reserve-locn: Environment → (Location × Environment)
  \[ \text{reserve-locn} = \lambda [\text{map}, l].[l, [\text{map}, \text{next-locn}(l)]] \]

The environment is a pair that also includes the last used location. This is so that further declarations can proceed from that point.

**Domain** Storable-value = Nat

**Domain** Store = Location → Storable-value

**Operations**

- access: Location → Store → Storable-value
  \[ \text{access} = \lambda l.\lambda s.s(l) \]

- update: Location → Storable-value → Store → Store
  \[ \text{update} = \lambda l.\lambda v.\lambda s.[l \mapsto v].s \]

**Domain** Poststore = OK + Err  
where OK = Err = Store

**Operations**

- return: Store → Poststore
  \[ \text{return} = \lambda s.\text{inOK}(s) \]

- signalerr: Store → Poststore
  \[ \text{signalerr} = \lambda s.\text{inErr}(s) \]
check: \((\text{Store} → \text{Poststore}_⊥) → (\text{Poststore}_⊥ → \text{Poststore}_⊥)\)

\[
\text{check } f = \lambda p.\cases{p \text{ of} \\
\text{isOK}(s) \rightarrow (f \ s) \\
\text{isErr}(s) \rightarrow p \\
\text{end}}
\]

The \text{check} operation has the job of looking at the store returned from executing a command and determining whether to proceed. A error can occur if an identifier is undeclared, resulting in an aborted assignment and an error store. The poststore is a lifted domain, and \text{check} is a strict function because of the possibility of an infinite loop.

**VALUATION FUNCTIONS**

Where we had functions of the store in the simple language without contexts, there will now be two arguments, the environment and the store. The environment will provide variable locations which will be constant for a particular block; the store will hold values which can change during the execution of the block. We should expect that any derivation will start out containing references to the environment, but these should disappear at “run-time”, just as they do in a real compiled language. This is what happens, as we shall see.

We start with the program, which is just a block. The initial environment, before any declarations, is empty.

\[
P: \text{Location} → \text{Store} → \text{Poststore}_⊥
\]

\[
P[K.] = \lambda l.\text{K}[K](\text{emptyenv } l)
\]

Note that this a function of a \text{Location} so that we can “relocate” the data to any starting location.

A block simply executes its command with an environment formed from the old one by adding the block’s declarations to it. The execution will be carried out in the store which it receives as a second argument.

\[
K: \text{Block} → \text{Environment} → \text{Store} → \text{Poststore}_⊥
\]

\[
K[\text{begin } D; \text{ end}] = \lambda e.\text{C}[C](\text{D}[D]e)
\]

Declarations are handled like commands, with function updates, except that they are stored in the environment. Constants are simply bound to the identifier. Variable identifiers are bound to a location which is obtained from the environment passed in. The location returned from \text{reserve-locn} is “one more” than the last-used location in the old environment.

\[
D: \text{Declaration} → \text{Environment} → \text{Environment}
\]

\[
D[D_1; D_2] = D[D_2] \circ D[D_1]
\]

\[
D[\text{const } I = N] = \text{updateenv } [I] \text{ in } \text{Nat} (N[N])
\]

\[
D[\text{var } I] = \lambda e.\text{let } [I', e'] = (\text{reserve-locn } e) \text{ in } \text{updateenv } [I] \text{ in } \text{Location} (I') e'
\]

The command sequence is handled by executing both in the same environment, but in different stores. This shows the major use of the environment: to give a context for command execution. \text{check} is used to look for undeclared identifiers.

\[
C: \text{Command} → \text{Environment} → \text{Store} → \text{Poststore}_⊥
\]

\[
C[C_1; C_2] = \lambda e. (\text{check } C[C_2]e) \circ (C[C_1]e)
\]

Assignment needs to return an error if either the left-hand variable is undeclared or the right-hand expression contains an undeclared identifier.
\[
C[I := E] = \lambda e.\lambda s. \text{cases } \text{accessenv}[I] e \text{ of } \\
\quad \text{isLocation}(l) \to (\text{cases } E[E] e s \text{ of } \\
\quad \quad \text{isNat}(n) \to (\text{return}(\text{update } l n s)) \\
\quad \quad \Diamond \text{isErrorvalue()} \to (\text{signalerr } s) \\
\quad \text{end}) \\
\quad \Diamond \text{isNat}(n) \to (\text{signalerr } s) \\
\quad \Diamond \text{isErrorvalue()} \to (\text{signalerr } s) \\
\text{end}
\]

Note the use of the operation \text{signalerr} to return an error poststore, and \text{return} to return the OK version.

The while loop is handled by our \text{fix} operator.

\[
C[\text{while } B \text{ do } C] = \lambda e.\lambda f.\lambda s. \text{cases } B[B] e s \text{ of } \\
\quad \text{isTr}(t) \to (t \to (\text{check } f) \circ (C[C] e) \Diamond \text{return})(s) \\
\quad \text{isErrorvalue()} \to (\text{signalerr } s) \\
\text{end}
\]

Operationally, this is rather inefficient, since the function is checked for bad declarations for each iteration. However, we know that if it passes the first time, it will pass every time.

Expression evaluation is very similar to the simpler language, except that we must return an error if an undeclared identifier is found. Note that if a location is found in the environment it must also map to a value in the store, or return an error, since the empty store maps all locations to error.

\[
E: \text{Expression} \to \text{Environment} \to \text{Store} \to \text{Expressible - value}
\]

\[
E[E_1 + E_2] = \lambda e.\lambda s. \text{cases } E[E_1] e s \text{ of } \\
\quad \text{isNat}(n_1) \to (\text{cases } E[E_2] e s \text{ of } \\
\quad \quad \text{isNat}(n_2) \to \text{inNat}(n_1 + n_2) \\
\quad \quad \Diamond \text{isErrorvalue()} \to \text{inErrorvalue()} \\
\quad \text{end} \\
\quad \Diamond \text{isErrorvalue()} \to \text{inErrorvalue()} \\
\text{end}
\]

\[
E[I] = \lambda e.\lambda s. \text{cases } (\text{accessenv}[I] e) \text{ of } \\
\quad \text{isNat}(n) \to \text{inNat}(n) \\
\quad \Diamond \text{isLocation}(l) \to \text{inNat}(\text{access } l s) \\
\quad \Diamond \text{isErrorvalue()} \to \text{inErrorvalue()} \\
\text{end}
\]

\[
E[N] = \lambda e.\lambda s. \text{inNat}(N[N])
\]

The valuation function \text{B} is the same as before. It has the functionality:

\[
B: \text{Environment} \to \text{Store} \to (\text{Tr} + \text{Errvalue})
\]
Consider the program

```
begin
  const A = 1;
  var X;
  X := A + 2;
  begin
    var A;
    while X = 0 do
      A := X
    end;
    X := A
  end.
end.
```

While this is not a terribly exciting program, it will illustrate how the nested context works and also reveal the compile-time/run-time distinction we are looking for.

First we annotate the program:

1. \(D_0 = \text{const } A = 1\)
2. \(D_1 = \text{var } X\)
3. \(C_1 = X := A + 2\)
4. \(C_2 = \text{begin } \text{var } A; \text{ while } A > 0 \text{ do } A := X \text{ end}\)
5. \(C_3 = X := A\)
6. \(C_4 = \text{while } A > 0 \text{ do } A := X\)
7. \(C_0 = C_1; C_2; C_3\)

The derivation starts:

\[P[D_0; D_1; C_0]\]

\[\lambda K[P[D_0; D_1; C_0]](\text{emptyenv } l)\]

Call the empty environment \(e_0\). Now applying \(K\) gives

\[C[C_0](D[D_0; D_1]e_0)\]

Working on the declarations first we have

\[D[D_0; D_1]e_0\]

\[= (D[\text{var } X] \circ D[\text{const } A = 1])e_0\]

\[= D[\text{var } X](\text{updateenv}[A] \circ \text{inNat}(one))e_0\]

\[= D[\text{var } X]e_1, \text{ where } e_1 = [\langle A \mapsto \text{inNat}(one) \rangle]e_0\]

Applying \(D\) again gives

\[D[\text{var } X]e_1\]

\[= \text{let } [l', e'] = \langle \text{reserve-locn } e_1 \rangle \text{ in}
  \text{updateenv}[X] \circ \text{inLocation}(l') e'\]

Let the location be \(l_1\) and the new environment with \(l_1\) be \(e_2\). When we update this with \(X\) we get environment \(e_3\). This is the environment for the execution of the program commands \(C_0\), i.e.

\[(\text{check } C[C_2; C_3]e_3) \circ (C[C_1]e_3)\]
We can work on these separately; the composition could then be applied to a store. However, for $C_1$, $C[X := A + 2]e_3$

$= \lambda s. \text{cases } \text{accessenv}[X]e_3 \text{ of }$

$\quad \text{isLocation}(l) \rightarrow (\text{cases } E[A + 2]e_3 s \text{ of }$

$\quad \quad \text{isNat}(n) \rightarrow (\text{return (update } l n s))$

$\quad \quad \text{isErrorvalue()} \rightarrow (\text{signalerr } s)$

$\quad \text{end})$

$\quad \text{isNat}(n) \rightarrow (\text{signalerr } s)$

$\quad \text{isErrorvalue()} \rightarrow (\text{signalerr } s)$

$\text{end}$

The result of accessing $X$ in the environment is $\text{inLocation}(l)$. Since this is a location, we evaluate the expression:

$E[A + 2]e_3$

$= \lambda s. \text{cases } E[A]e_3 s \text{ of }$

$\quad \text{isNat}(n_1) \rightarrow (\text{cases } E[2]e_3 s \text{ of }$

$\quad \quad \text{isNat}(n_2) \rightarrow \text{inNat}(n_1 \text{ plus } n_2)$

$\quad \quad \text{isErrorvalue()} \rightarrow \text{inErrorvalue()}$

$\quad \text{end})$

$\quad \text{isErrorvalue()} \rightarrow \text{inErrorvalue}()$

$\text{end}$

$= \lambda s. \text{inNat(} \text{one plus two})$

$= \lambda s. \text{inNat(three)}$

So the assignment results in $\lambda s. \text{return (update } l \text{ three } s)$

So the composition now is

$(\text{check } C[C_2; C_3]e_3) \circ (\lambda s. \text{return (update } l \text{ three } s))$

Working on the second two commands, we get:

$(\text{check } C[C_3]e_3) \circ (C[\text{begin var } A; C_4 \text{end}]e_3)$

The nested block gives us $C[C_4](D[\text{var } A]e_4)$

Declaring the variable gives us $e_5$, i.e. the update of $e_4$, which is $e_3$ with a new location, $l_2$.

$e_5 = \text{updateenv}[A] \text{inLocation}(l_2) e_4$

The loop is thus executed with environment $e_5$. This is:
\[ C[\textbf{while } X = 0 \textbf{ do } A := X] e_5 = \text{fix}(\lambda f. \lambda s. \text{cases } B[X = 0] e_5 s \text{ of}
\]
\[ \text{isTr}(t) \rightarrow (t \rightarrow (\text{check } f) \circ (C[A := X] e_5) \text{ return}) s
\]
\[ \text{isErrorvalue}() \rightarrow (\text{signalerr } s)
\]
\[ \text{end}
\]

Since \( X \) is bound to \( l_1 \) in \( e_5 \), the evaluation of \( X = 0 \) gives us some truth value. We don’t know which because we don’t have a store, but we can simplify the \textit{fix} expression to

\[ \text{fix}\left(\lambda f. \lambda s. (\text{access } l_1 s) \text{ equals zero } \rightarrow (\text{check } f) \circ (C[A := X] e_5) \text{ return}\right) s \]

Applying \( C \) to \( A := X \) give

\[ C[A := X] e_5 = \lambda s. \text{cases accessenv}[A] e_5 \text{ of}
\]
\[ \text{isLocation}(l) \rightarrow (\text{cases } E[X] e_5 s \text{ of}
\]
\[ \text{isNat}(n) \rightarrow (\text{return } (\text{update } l \ n \ s))
\]
\[ \text{isErrvalue}() \rightarrow (\text{signalerr } s)
\]
\[ \text{end}
\]
\[ \text{isNat}(n) \rightarrow (\text{signalerr } s)
\]
\[ \text{isErrvalue}() \rightarrow (\text{signalerr } s)
\]
\[ \text{end}
\]

Since accessing \( A \) in \( e_5 \) gives us \( l_2 \), and evaluating \( X \) in \( e_5 \) gives will give us a \( \text{Nat} \), this simplifies to

\[ \lambda s. \text{return } (\text{update } l_2 (\text{access } l_1 \ s) s) \]

If we further simplify the last assignment we get

\[ C[X := A] e_5 = \lambda s. \text{cases accessenv}[X] e_5 \text{ of}
\]
\[ \text{isLocation}(l) \rightarrow (\text{cases } E[A] e_5 s \text{ of}
\]
\[ \text{isNat}(n) \rightarrow (\text{return } (\text{update } l \ n \ s))
\]
\[ \text{isErrvalue}() \rightarrow (\text{signalerr } s)
\]
\[ \text{end}
\]
\[ \text{isNat}(n) \rightarrow (\text{signalerr } s)
\]
\[ \text{isErrvalue}() \rightarrow (\text{signalerr } s)
\]
\[ \text{end}
\]

This simplifies to

\[ \lambda s. \text{return } (\text{update } l \ \text{one } s) \]

If we assemble all the parts we get this large expression

\[ (\text{check } \lambda s. \text{return } (\text{update } l \ \text{one } s)) \circ
\]
\[ \left( \text{fix}\left(\lambda f. \lambda s. (\text{access } l_1 s) \text{ equals zero } \rightarrow (\text{check } f) \circ (\text{update } l_2 (\text{access } l_1 s) s) \text{ return}\right) s \right) \circ
\]
\[ (\lambda s. \text{return } (\text{update } l \ \text{three } s)) \]

If we replace \( l_2 \) by \( (\text{next-locn } l_1) \), and the original \( \lambda l \) from the \( P \) function, we get
\[
\lambda. \left( \text{check} \left( \text{check} \lambda.s. \text{return} (\text{update} \ l \ s) \right) \right) \circ \\
\lambda. \left( \text{fix} \left( \left( \lambda f \lambda.s. \left( \text{access} \ l \ s \right) \text{equals} \ zero \to (\text{check} \ f) \circ \right) \right) \right) \circ \\
(\lambda.s. \text{return} (\text{update} \ l \ three \ s))
\]

If we use the equivalence
\[
(\text{check} (\text{check} \ f \circ g)) = (\text{check} \ f) \circ (\text{check} \ g),
\]
then we can write
\[
\lambda. (\text{check} \ \lambda.s. \text{return} (\text{update} \ l \ one \ s)) \circ \\
\lambda. \left( \text{check} \left( \text{fix} \left( \left( \lambda f \lambda.s. \left( \text{access} \ l \ s \right) \text{equals} \ zero \to (\text{check} \ f) \circ \right) \right) \right) \circ \\
(\lambda.s. \text{return} (\text{update} \ l \ three \ s))
\]

We can even reverse the compositions so that the expression reads forwards like the program:
\[
\lambda. (\text{check} \ \lambda.s. \text{return} (\text{update} \ l \ three \ s)) ! \\
\lambda. \left( \text{check} \left( \text{fix} \left( \left( \lambda f \lambda.s. \left( \text{access} \ l \ s \right) \text{equals} \ zero \to (\text{update} \ \text{next-locn} \ l) (\text{access} \ l_2 \ s) \right) \circ \text{return} \right) \right) \circ \\
(\text{check} \ \lambda.s. \text{return} (\text{update} \ l \ one \ s))
\]

Where \( f ! g = g \circ f \).

Whichever way we write it, two facts are clear. The first is that all the environments have disappeared, leaving only locations in stores. Second is that this expression resembles a compiled program, containing only machine-like instructions (the semantic operations). If we “run” the program with an initial location and an empty store, we will produce a final store with both locations (for X in the outer block and A in the inner) still in place. This is not quite the stack discipline we are used to in block-structured languages, but this can be added with suitable changes in the semantic algebras.

Note that the valuation function for the sequence of commands ensures that the same environment is used for both. If both commands are blocks, any new locations allocated in the first block are “unallocated” for the second block. So the program:

\[
\begin{align*}
\text{begin} \\
\text{begin} \\
\quad \text{var} \ x; \\
\quad C_1 \\
\text{end} \\
\text{begin} \\
\quad \text{var} \ x; \\
\quad C_2 \\
\text{end}
\end{align*}
\]

will execute with the empty environment passed to both the first block and the second block. Each block augments the environment passed to it with a declaration for x using the next-highest location stored with the environment. It will be the same location as allocated for the second block.