A LANGUAGE WITH TYPE-CHECKING

To introduce the idea behind type checking, we will change our domain of values to one that includes two types. Without introducing a brand new domain, we will simply use a sum domain formed from the truth values and the natural numbers. The new domain is:

\[ \text{Storeable-value} = \text{Tr} + \text{Nat} \]

Values in this domain will be tagged as either a truth value, or a number. If we use natural numbers as the tags, then elements of this set are, e.g. \([0, \text{true}]\) and \([1, \text{zero}]\).

As soon as we do this, however, our language becomes an order of magnitude more difficult to handle. There are three things we will need to change.

1. expression evaluation. When we evaluate a binary operation, the two operands will have to be checked for compatibility. We cannot, for instance, evaluate \(\text{one plus true}\), even though the program that contains this might be perfectly well-formed:
   \[
   x := \text{true}; \\
y := 1 + x
   \]
   (Our language will be dynamically-typed, i.e. there will be no declarations.) However, expressions such as \(\text{one plus true}\) will have to take a value, so we will augment our domains with a domain that includes an error value:

\[ \text{Expressible-value} = \text{Storeable-value} + \text{Errvalue} \]

Where, \(\text{Errvalue}\) is just the \(\text{Unit}\) domain that corresponds to the empty set.

2. assignment. Since any expression, can have an error value, we need to prevent its assignment in, \(y := 1 + x\), above.

3. command sequence. Finally, if an abortive assignment has been executed, there is no sense carrying on, so the sequence

\[
y := 1 + x; \\
z := y + 2
\]

should be prevented.

We could handle all of these problems in the valuation equations. However, it makes more sense to generalize the checking of error situations by placing most of the effort in the semantic algebras.

ABSTRACT SYNTAX

This language is only a slight modification from the previous one. The biggest change is that true and false are values which can be assigned to identifiers.

\[
P \in \text{Program} \\
C \in \text{Command} \\
E \in \text{Expression} \\
I \in \text{Id} \\
N \in \text{Numeral}
\]

\[
P ::= C \\
C ::= C_1; C_2 | I := E | \text{if } E \text{ then } C_1 \text{ else } C_2 | \text{diverge} \\
E ::= E_1 + E_2 | E_1 = E_2 | \neg E | (E) | I | N | \text{true} | \text{false}
\]

SEMANTIC ALGEBRAS FOR A LANGUAGE WITH TYPES

Starting from the value domain, as above:
Domain $\text{Storeable-value} = Tr + Nat$

We will add the error value and a way to check whether one half of a binary expression is in error, so we need not evaluate the second half.

Domain $\text{Expressible-value} = \text{Storeable-value} + \text{Errvalue}$
where $\text{Errvalue} = \text{Unit}$

Operations

$\text{check-expr} : (\text{Store} \to \text{Expressible-value}) \times (\text{Storeable-value} \to \text{Store} \to \text{Expressible-value})$

$\to (\text{Store} \to \text{Expressible-value})$

The operation has a functionality that has two arguments, both functions. The first function will be a function that takes a store and returns a value, possibly an error value. The second function will take that value, (if it is not an error), and the value from another evaluation, and if the second one is not in error, will perform the actual binary operation. This actual operation will be part of the valuation function. This domain operation just provides the framework. Its definition is expressed as an infix operation:

$f_1 \text{check-expr} f_2 = \lambda s. \text{cases} (f_1 s)$ of

- is $\text{Storeable-value}(v) \to (f_2 v s)$
- $\square$ is $\text{Errvalue}() \to \text{inErrvalue}()$

end

The cases form is used to distinguish between a good value and an error value. Only if the first function produces a good value (with the $\text{Storeable-value}$ tag) will the second function be applied. This sequencing is clearly important in this kind of checking, and the technique of writing it as an infix operation makes this sequencing very clear.

When we move to look at the assignment problem, we must bring the store into the picture. We already know about the possibility of a “non-terminating” store, which results form executing $\text{diverge}$, but now we need also to add the possibility of an “error” store. We will thus create a new domain called $\text{Post-store}$ to represent this:

Domain $\text{Store} = Id \to \text{Storeable-value}$

Operations as before, except replace $\text{Nat}$ with $\text{Storeable-value}$.

Domain $\text{Post-state} = \text{OK} + \text{Err}$
where $\text{OK}$ and $\text{Err} = \text{Store}$

Operations

$\text{check-result}: (\text{Store} \to \text{Expressible-value}) \times (\text{Storeable-value} \to \text{Store} \to \text{Post-store}_\bot)$

$\to (\text{Store} \to \text{Post-store}_\bot)$

This operation is similar to $\text{check-expr}$, but instead of sequencing two expression evaluations, it sequences an expression evaluation and a command execution. The first argument will be a function that returns a value, which could be an error value. If it is not, then the function from the second argument will be applied which will execute a command. If the value is an error value, then the command will be skipped. Note that the final store is lifted to accommodate the possibility of non-termination.

$f \text{check-result} g = \lambda s. \text{cases} (f s)$ of

- is $\text{Storeable-value}(v) \to (g v s)$
- $\square$ is $\text{Errvalue}() \to \text{inErr}(s)$

end

The last operation is the one that checks the sequencing of two commands.

$\text{check-cmd}: (\text{Store} \to \text{Post-store}_\bot) \times (\text{Store} \to \text{Post-store}_\bot) \to (\text{Store} \to \text{Post-store}_\bot)$

The first function will execute the first command in the sequence. If this returns an ‘OK’ store, then the second function will execute the second command. If the first one fails, the second will not be executed.
\[ h_1 \text{check-cmd} \quad h_2 = \lambda a. \text{let } z = (h_1 \ a) \text{ in } \]
\[ \begin{align*}
\text{cases } z \text{ of } \\
& \text{isOK}(s) \rightarrow (h_2 \ s) \\
& \Box \text{isErr}(s) \rightarrow z
\end{align*}
\]

\section*{VALUATION FUNCTIONS}

**P:** Program $\rightarrow$ Store $\rightarrow$ Post-store

\[ P[C] = \lambda s. C[C] s \]

**C:** Command $\rightarrow$ Store $\rightarrow$ Post-store

\[ C[C_1; C_2] = C[C_1] \text{check-cmd } C[C_2] \]

\[ C[I := E] = E[E] \text{check-result} \left( \lambda v. \lambda s. \text{inOK} \left( \text{update}[I] \ v \ s \right) \right) \]

The second function passed to check-result does the actual update of the store if the evaluation of E returns a valid value. If E returns an error value, the second function is not applied, and the update is not done.

\[ C[\text{if } E \text{ then } C_1 \text{ else } C_2] = E[E] \text{check-result} \left( \lambda s. \lambda v. \text{cases } v \text{ of } \right. \\
\left. \begin{align*}
& \text{isTr}(t) \rightarrow \left( t \rightarrow C[C_1] \Box C[C_2] \right) s \\
& \Box \text{isNat}(n) \rightarrow \text{inErr}(s)
\end{align*} \right) \]

Here the second function tests the type of result of evaluating the expression. If it is a number, then an error is signaled. If it is Boolean, however, then its value is used to choose between executing \( C_1 \) and \( C_2 \).

\[ C[\text{diverge}] = \lambda s. \bot \]

This is the same as before.

**E:** Expression $\rightarrow$ Store $\rightarrow$ Expressible-value

\[ E[E_1 + E_2] = E[E_1] \text{check-expr} \]

\[ \left( \lambda v. \text{cases } v \text{ of } \right. \\
\left. \begin{align*}
& \text{isTr}(t) \rightarrow \text{inErrvalue}() \\
& \Box \text{isNat}(n) \rightarrow E[E_2] \text{check-expr} \left( \lambda v'. \lambda s. \text{cases } v' \text{ of } \right. \\
\left. & \text{isTr}(t') \rightarrow \text{inErrvalue}() \\
& \Box \text{isNat}(n) \rightarrow \text{inStoreable-value}(\text{inNat}(n \ \text{plus} \ n'))
\end{align*} \right) \]

The addition can only succeed if both expression evaluate to a number. The second function test the result of evaluating the first expression. Only when it is a number do we evaluate the second expression. check-expr is used again to test the type of its result. Notice the two tags put on the result \( n \ \text{plus} \ n' \). The inNat tag makes it a Storeable-value (to distinguish it from a Boolean). The inStoreable-value tag makes it an Expressible-value, and distinguishes it from an Errvalue.

\[ E[E_1 = E_2] = E[E_2] \text{check-expr} \]

\( \left( \lambda v. \text{cases } v \text{ of } \right. \\
\left. \begin{align*}
& \text{isTr}(t) \rightarrow \text{inErrvalue}() \\
& \Box \text{isNat}(n) \rightarrow \text{inStoreable-value}(\text{inNat}(n \ \text{plus} \ n'))
\end{align*} \right) \]
isTr(t) \rightarrow \lambda s.\text{inErrvalue()}

\text{\textup{isNat}(n) \rightarrow \text{E}[E_2]\text{-check-expr}}

(\lambda v'.\lambda s.\text{cases } v' \text{ of }
\text{isTr}(t') \rightarrow \text{inErrvalue()}
\text{\textup{isNat}(n) \rightarrow \text{inStoreable-value}(\text{inTr}(n \text{ equals } n'))}
\text{end})
\text{end)}

This is identical to the addition, except that the \textit{Tr} tag is put on the result of carrying the \textit{equals} test.

\text{E[\sim E] = E[E]\text{-check-expr}}

(\lambda v.\lambda s.\text{cases } v \text{ of }
\text{isTr}(t) \rightarrow \text{inStoreable-value}(\text{inTr}(\text{not } t))
\text{\textup{isNat}(n) \rightarrow \text{inErrvalue()}}
\text{end})
\text{end)}

There is no second expression here, so all the work is done in the second function which ignores the \textit{Store} passed in.

\text{E[(E)] = E[E]}

\text{E[I] = \lambda s.\text{inStoreable-value}(\text{access}[1].s)}

\text{E[N] = \lambda s.\text{inStoreable-value}(\text{inNat}(N[N]))}

\text{E[true] = \lambda s.\text{inStoreable-value}(\text{inTr}(true))}

\text{E[false] = \lambda s.\text{inStoreable-value}(\text{inTr}(false))}

These are all the same as before, except that the \textit{Storeable-value} tag is put on the value, which is tagged as a Boolean or a number.

Rather than use these infix operators, we could write the valuation functions more straightforwardly. For example, instead of using \textit{check-expr} in the function for \texttt{+}, we could write:
\[ C[E_1 + E_2] = \]
\[ \lambda x. \text{let } e_1 = E[E_1] \text{ in cases } e_1 \text{ of isErrorvalue() \rightarrow inErrorvalue()}
\]
\[ \square \text{ isStorable-value}(v_1) \rightarrow cases v_1 \text{ of isTr}(t) \rightarrow inErrorvalue()
\]
\[ \square \text{ isNat}(n_1) \rightarrow \text{ let } e_2 = E[E_2] \text{ in cases } e_2 \text{ of isErrorvalue() \rightarrow inErrorvalue()}
\]
\[ \square \text{ isStoreable-value}(v_2) \rightarrow cases v_2 \text{ of isTr}(t) \rightarrow inErrorvalue()
\]
\[ \square \text{ isNat}(n_2) \rightarrow \text{ inStorable-value(inNat(n_1 \text{ plus } n_2)) end end end end end end end end}
\]

Using `checkExpr` reduces the levels of nesting at the expense of making the operation harder to understand.

**EXAMPLE DERIVATION**

To illustrate the type checking, let us carry out the derivation of the following program:

\[ x := 1; x := \text{true}; \]
\[ \text{if } x + 1 = 0 \text{ then diverge else } x := 0 \]

Note that x gets values of different types. This is OK because the language is dynamically typed. However, the expression \( x + 1 \) should give us trouble, and be trapped as an error.

\[ P[x := 1; x := \text{true}; \text{if } x + 1 = 0 \text{ then diverge else } x := 0](\text{newstore}) \]
\[ = (\lambda x. C[x := 1; x := \text{true}; \text{if } x + 1 = 0 \text{ then diverge else } x := 0](\text{newstore}) \]
\[ = C[x := 1; x := \text{true}; \text{if } x + 1 = 0 \text{ then diverge else } x := 0](\text{newstore}) \]
\[ = (C[x := 1]\check\text{-cmd } C[x := \text{true}; \text{if } x + 1 = 0 \text{ then diverge else } x := 0])(\text{newstore}) \]

If we expand `check-cmd`, we get
let \( z = C[x := 1](\text{newstore}) \) in
\[
\begin{array}{l}
cases z of \\
is\text{OK}(s) \rightarrow C[x := \text{true}; if x + 1 = 0 then \text{diverge} else x := 0](s) \\
\quad \triangleright is\text{Err}(s) \rightarrow z \\
end
\end{array}
\]

Executing \( x := 1 \) gives:
\[
C[x := 1](\text{newstore})
\]
\[
= (E[1]\text{ check - result } (\lambda v.\lambda s.\text{in\text{OK}}(\text{update}[x]v s)))\text{newstore}
\]

Expanding \text{check-result} gives:
\[
\begin{array}{l}
cases E[1]\text{newstore} of \\
is\text{Storeable-value}(v) \rightarrow \text{in\text{OK}}(\text{update}[x]v \text{newstore}) \\
\quad \triangleright is\text{Err\text{value}()} \rightarrow \text{in\text{Err}}(\text{newstore}) \\
end
\end{array}
\]

Since \( E[1]\) gives \text{in\text{Storeable-value}(in\text{Nat}(\text{one}))}, \) which has the \text{Storeable-value} tag, this reduces to
\[
\text{in\text{OK}}(\text{update}[x] \text{in\text{Nat}(\text{one})} \text{newstore})
\]

which is the store \( [x] \mapsto \text{in\text{Nat}(\text{one})} \text{newstore} \) with an OK tag. Call this store \( s_1 \). Going back to the let, since the OK tag is present on the store, we execute:
\[
C[x := \text{true}; if x + 1 = 0 then \text{diverge} else x := 0](s_1)
\]

Now the \text{check-cmd} expansion gives:
\[
\begin{array}{l}
let z = C[x := \text{true}](s_1) \text{ in} \\
cases z of \\
is\text{OK}(s) \rightarrow C[if x + 1 = 0 then \text{diverge} else x := 0](s) \\
\quad \triangleright is\text{Err}(s) \rightarrow z \\
end
\end{array}
\]

and executing the assignment \( x := \text{true} \) gives a store \( s_2 = [x] \mapsto \text{in\text{Tr}(true)}\)([x] \mapsto \text{in\text{Nat}(\text{one})} \text{newstore}),\) which reduces to \( [x] \mapsto \text{in\text{Tr}(true)} \text{newstore} \) using extensionality of functions. This is the store for executing the conditional:
\[
C[if x + 1 = 0 then \text{diverge} else x := 0](s_2)
\]
\[
= (E[x + 1 = 0] \text{ check - result } (\lambda s.\lambda v.\text{cases } v \text{ of} \\
is\text{Tr}(t) \rightarrow (t \rightarrow C[\text{diverge} \triangleright C[x := 0]])(s) \\
\quad \triangleright is\text{Nat}(n) \rightarrow \text{in\text{Err}(s)} \\
end))(s_2)
\]

Expanding \text{check-result} gives
cases \((E\left[x + 1 = 0\right]_{s_2})\) of

\[\text{isStorable\text{-}value}(v) \rightarrow \text{(cases } v \text{ of} \]
\[\text{isTr}(t) \rightarrow \left(t \rightarrow C\left[\text{diverge}\right] \sqcap C\left[x := 0\right]\right)_{s_2} \]
\[\sqcap \text{isNat}(n) \rightarrow \text{inErr}(s_2) \]
\[\text{end}) \]
\[\text{isErrvalue()} \rightarrow \text{inErr}(s_2) \]
\end{cases}
end

We need to evaluate \(x + 1 = 0\) in \(s_2\) to find out whether the conditional can go through. This derivation is \((E\left[x + 1\right]_{s_2})\check\text{-expr}

\[\text{cases } E\left[x + 1\right]_{s_2} \text{ of} \]
\[\text{isStorable\text{-}value}(v) \rightarrow \text{(cases } v \text{ of} \]
\[\text{isTr}(t) \rightarrow \lambda s.\text{inErrvalue()} \]
\[\sqcap \text{isNat}(n) \rightarrow E\left[0\right]_{s_2}\check\text{-expr} \]
\[\text{(}\lambda v'.\lambda s.\text{cases } v' \text{ of} \]
\[\text{isTr}(t') \rightarrow \text{inErrvalue()} \]
\[\sqcap \text{isNat}(n) \rightarrow \text{inStorable\text{-}value}(\text{inTr}(n \text{ equals } n')) \]
\[\text{end})_{s_2} \]
\[\text{end})_{s_2} \]

Expanding the top-level \(check\text{-expr}\) we get

\[\text{cases } E\left[x + 1\right]_{s_2} \text{ of} \]
\[\text{isStorable\text{-}value}(v) \rightarrow \text{(cases } v \text{ of} \]
\[\text{isTr}(t) \rightarrow \lambda s.\text{inErrvalue()} \]
\[\sqcap \text{isNat}(n) \rightarrow E\left[0\right]_{s_2}\check\text{-expr} \]
\[\text{(}\lambda v'.\lambda s.\text{cases } v' \text{ of} \]
\[\text{isTr}(t') \rightarrow \text{inErrvalue()} \]
\[\sqcap \text{isNat}(n) \rightarrow \text{inStorable\text{-}value}(\text{inTr}(n \text{ equals } n')) \]
\[\text{end})_{s_2} \]
\[\text{end})_{s_2} \]

Working on \(E\left[x + 1\right]_{s_2}\) we get
The evaluation of \(x\) gives 
\[\text{inStorable-value(access}[x]s_2), \text{ which is} \]
\[\text{inStorable-value(inTr(true))}\]
Thus the Storable-value tag is present, which means the evaluation of the constant 1 is skipped, and the function \(\lambda s.\text{inErrvalue()}\) is applied to \(s_2\) to produce the value \(\text{inErrvalue()}\). This means that \(\text{E}[x + 1]s_2\) has the same value, and so does \(\text{E}[x + 1 = 0]s_2\). This means that, from the expansion of \text{check-result} in the execution of the conditional the store will be tagged as an error store, i.e. \(\text{inErr}(s_2)\). This is the denotation of the whole program. If there were extra commands after this conditional, they would be skipped after the checking in \text{check-cmd}.

**ADDION OF INPUT-OUTPUT**

Out language can be augmented with real input-output with the addition of two domains, and modifications to the valuation functions. The two domains are Input and Output:

Domain \(\text{Input} = \text{Expressible-value}^{*}\)

Operations

\(\text{get-value}: \text{Input} \rightarrow (\text{Expressible-value} \times \text{Input})\)

\(\text{get-value} = \lambda i.\text{null} i \rightarrow (\text{inErrvalue()}, i) \sqcup (\text{hd} i, \text{tl} i)\)

Domain \(\text{Output} = (\text{Storable-value} + \text{String})^{*}\)

Operations
empty: Output
empty = nil

put-value: Storeable-value × Output → Output
put-value = λ(v, o).in Storeable-value(v) cons o

put-message: String × Output → Output
put-message = λ(t, o).in String(t) cons o

Both of these new domains are represented as list of values, or streams. The input domain includes the possibility of reading from an empty stream, so it is Expressible-values. The output stream can accept either Storeable-values or values form a primitive domain of strings. This form can be used for error messages. The basic manipulations of the streams is done with the list operations hd, tl, and cons.

Instead of a simple store, we need to add the input and output streams into a State:

Domain State = Store × Input × Output

This will replace the store arguments in all the valuation functions that need them. Where we need to break up the state into its components, we will do this with a triple [s, i, o] of store, input and output streams. For instance the valuation function for assignment becomes:

$$C[1 := E] = E[E] \text{ check-result}(\lambda v.\lambda [s,i,o].\text{inOK}((\text{update}[1]v s),i,o))$$

Two new commands for read and write have the following semantics:

$$C[\text{read } 1] = \lambda [s,i,o].\text{let } [x,i'] = \text{get-value}(i) \text{ in }$$
$$\text{cases } x \text{ of }$$
$$\text{isStoreable-value}(v) \rightarrow \text{inOK}((\text{update}[1]v s),i,o)$$
$$\text{isErrvalue()} \rightarrow \text{inErr}(s,i', \text{put-message}("bad input", o))$$
$$\text{end}$$

So reading a value from the input stream is like assignment, except the value does not come from an expression.

$$C[\text{write } E] = E[E] \text{ check-result}(\lambda v.\lambda [s,i,o].\text{inOK}(s,i,\text{put-value}(v,o)))$$

Since the expression whose value is to be written could be in error, we use check-result before actually doing the output operation.

We can get the type-checking operations to write error messages into the output stream. E.g.

$$f \text{ check-result } g = \lambda [s,i,o].\text{cases } (f s) \text{ of }$$
$$\text{isStoreable-value}(v) \rightarrow (g v [s,i,o])$$
$$\text{isErrvalue()} \rightarrow \text{inErr}(s, i, \text{put-message}("type error"), o))$$
$$\text{end}$$