Course Notes: Formalising exceptions

These notes consist of two short sections. The first discusses how the notion of exceptions and exception handling can be introduced into our language of arithmetic expressions, Exp. The second then considers its addition to the language of imperative language of commands, While.

1 Exceptions in expressions: big-step semantics

\[ E \in \text{Arith}^{\text{exc}} ::= l \in \text{Locs} \mid n \in \text{Nums} \mid (E + E) \]
\[ \mid E - E \mid E \div E \]
\[ \mid \text{throw } n \mid \text{try } E \text{ catch } l \text{ with } E \]

Figure 1: The language Arith^{exc}

A language of arithmetic expressions which can raise and catch expressions is given Figure 1. The intuition behind the language is as follows:

1. Evaluation of expressions require a store. For example the value of \( l_1 + 2 \) depends on the current value stored in the location \( l_1 \).

2. For the purpose of illustration we will assume that the subtraction and division operators will raise exceptions if the values to which they are applied are not appropriate.

3. The default evaluation of expressions should proceed as standard, unless exceptions arise.

This will be formalised (in the standard way) as judgements of the form

\[ \langle E, s \rangle \Downarrow \langle n, s' \rangle \]  

signifying: The default evaluation of \( E \) relative to the store \( s \) should result in the numeral \( n \); the side-effect of this evaluation is to transform the store \( s \) into the store \( s' \). So for example let \( s \) be the (standard) store in which the numeral \( k \) is stored in the location \( l_k \). Then we would expect:
The rules defining the default judgements (1) are given in Figure 2. The rules for the subtraction and division operators are straightforward, provided the operands evaluate to untroublesome values. The rule for the try expression is also straightforward; if the evaluation of \( E_1 \) proceeds normally then the catch mechanism is ignored.

One can check that with these rules the judgements (i) and (ii) above can be derived.

5. To make the language somewhat interesting let us assume that \( E_1 - E_2 \) raises an exception whenever the result of evaluating \( E_2 \) is larger than the result of evaluating \( E_1 \). Similarly for \( E_1 \div E_2 \) when the former is not an exact divisor of the latter.

6. To make the semantics clearer we include an explicit expression to indicate that an exception has been raised, \( \text{throw } n \). It is standard to have exceptions of different kinds in a language; in ours we use numerals for this purpose. As a design choice (fairly arbitrary) let us say a misuse of subtraction raises an exception of value 0 while misuse of \( \div \) raises an exception of value 1.
7. Intuitively when \( \text{try } E_1 \text{ catch } l \text{ with } E_2 \) is evaluated, if an exception of value \( k \) is raised when evaluating \( E_1 \) then

- the value \( k \) is stored in location \( l \)
- \( E_2 \) is then evaluated relative to the modified store.

Intuitively the exception is caught in location \( l \) and the alternative evaluation of \( E_2 \) is launched.

So for example we would expect:

(i) \( \langle \text{try} (l_1 - l_3) \text{ catch } l_6 \text{ with } (l_7 + l_6), s \rangle \Downarrow \langle 7, s[l_6 \mapsto 0] \rangle \)

(ii) \( \langle \text{try} (l_8 \div l_3) \text{ catch } l_6 \text{ with } (l_7 + l_6), s \rangle \Downarrow \langle 8, s[l_6, l_7 \mapsto 1, 8] \rangle \)

In order to give rules for inferring consequences such as (i) and (ii) above the standard judgements, as in (1) above, are no longer sufficient. In order to explain the behaviour of \( \text{try } E_1 \text{ catch } l \text{ with } E_2 \) we need to be able to express the possible exceptions that \( E_1 \) can raise. For this purpose we introduce new judgements

\[ \langle E_1, s \rangle \Downarrow \langle \text{throw } n, s' \rangle \]

indicating that when \( E \) is evaluated relative to the store \( s \) the exception \( \text{throw } n \) is raised, and as a side-effect the store is changed to \( s' \). So for example we would expect to be able to infer from our rules, yet to be introduced, the following judgements:

(i) \( \langle l_8 \div l_3, s \rangle \Downarrow \langle \text{throw } 1, s \rangle \)

(ii) \( \langle \text{try} (l_8 \div l_3) \text{ catch } l_6 \text{ with } (l_6 - l_7), s \rangle \Downarrow \langle \text{throw } 0, s[l_6 \mapsto 1] \rangle \)

There are two aspects to the organisation of exceptions:

- how they are initially occur
- how they are propagated through expressions.

The rules governing the \textit{throwing} of exceptions are given in Figure 3. They reflect the informal English explanations already given above; \textit{subtraction} and division raise the exceptions \( 0, 1 \), respectively, as decided upon, if their arguments turn out not to be suitable.

The rules governing the propagation of exceptions are given in Figure 4, with the most important rule being \((b\cdot\text{TRY})\). In fact this encompasses two rules, which we
The expressions for throwing exceptions are given by

\[(b\text{-}\text{MINUS}E)\]
\[
\langle E_1, s \rangle \downarrow \langle n_1, s_1 \rangle \quad \langle E_2, s_1 \rangle \downarrow \langle n_2, s_2 \rangle \quad n_2 > n_1
\]

\[(b\text{-}\text{DIV}E)\]
\[
\langle E_1, s \rangle \downarrow \langle n_1, s_1 \rangle \quad \langle E_2, s_1 \rangle \downarrow \langle n_2, s_2 \rangle \quad \exists k \in \mathbb{N}. k \cdot n_2 = n_1
\]

\[(b\text{-}\text{THROW}E)\]
\[
\langle \text{throw} n, s \rangle \downarrow \langle \text{throw} n, s \rangle
\]

Figure 3: Expressions: throwing exceptions

The expressions for propagating exceptions are given by

\[(b\text{-}\text{TRY}P)\]
\[
\langle E_1, s \rangle \downarrow \langle \text{throw} n_1, s_1 \rangle \quad \langle E_2, s_1[l \mapsto n_1] \rangle \downarrow \langle t, s_2 \rangle
\]

\[\langle \text{try} E_1 \text{ catch } l \text{ with } E_2, s \rangle \downarrow \langle t, s_2 \rangle\]

\[(b\text{-}\text{THROW}P)\]
\[
\langle E, s \rangle \downarrow \langle \text{throw} n, s' \rangle
\]

\[\langle \text{throw} E, s \rangle \downarrow \langle \text{throw} n, s' \rangle\]

\[(b\text{-}\text{OP}P)\]
\[
\langle E_1, s \rangle \downarrow \langle n_1, s_1 \rangle \quad \langle E_2, s_1 \rangle \downarrow \langle \text{throw} n_2, s_2 \rangle
\]

\[\langle E_1 \text{ op } E_2, s \rangle \downarrow \langle \text{throw} n_2, s_2 \rangle\]

\[\langle E_1, s \rangle \downarrow \langle \text{throw} n_1, s_1 \rangle\]

\[\langle E_1 \text{ op } E_2, s \rangle \downarrow \langle \text{throw} n_1, s_1 \rangle\]

Figure 4: Expressions: propagating exceptions

have reduced to a single rule by introducing some meta-notation. Here we use the meta-variable \(t\) which can stand for either a value \(n\), the result of a successful evaluation, or an exception throw \(n\). Then, with this rule, the result of evaluating try \(E_1\) catch \(l\) with \(E_2\) relative to a store \(s\), if \(E\) raises an exception, is determined by the evaluation of the catch expression \(E_2\) relative to the store in which the location \(l\) contains the value of the exception.
However the other propagation rules are also important. For example we would expect `throw (2 - 7)` to raise the `subtraction` exception `throw 0` and for `throw (7 ÷ 2)` to raise the exception `throw 1`, the division exception; this is the import of the rule `(subs-throw,p)`. What about the evaluation of `(2 - 7) + (7 ÷ 2)`? An exception should be raised but which? This is a design decision of the semantics. We could allow the possibility of either exception being raised, or we could dictate a particular choice. The latter is the effect of the two rules `(subs-op,p)`, where we use `op` to represent any of our operator symbols. In effect these hint at a left-to-right evaluation of expressions. For example evaluation of the above expression leads to the raising of the subtraction exception `throw 0`.

**Exercise 1** Use the big-step semantics to determine the result of evaluating the following expressions relative to the standard store. In each case you should give a derivation of your answer using the big-step rules.

(i) `try (l_2 + l_3) catch l_4 with (l_4 - l_4)`

(ii) `try (l_2 ÷ l_3) catch l_0 with (l_7 + l_0)`

(iii) `try (l_8 ÷ l_3) catch l_6 with (l_6 - l_3)`

(iv)

```
try (try (l_8 ÷ l_3) catch l_6 with (l_6 - l_3))
catch l_4
with (l_4 + l_6)
```

(v)

```
try (l_8 ÷ l_3)
catch l_4
with (try (l_6 - l_3) catch l_4 with (l_4 + l_6))
```

**Exercise 2** Consider the following statement:

For every $E \in \text{Arith}^\text{exc}$, if $\tau_{\text{big}} \langle E, s_i \rangle \downarrow \langle n, s_t \rangle$ then the terminal store $s_t$ coincides with the initial store $s_i$.

Is it true or false? If it is true give a proof. If it is false give a counter-example.
Exercise 3  Many programming languages which have constructs for raising and catching exceptions also contain a construct ... and finally ... for tidying up at the end of the execution of a program.

Extend the language Arithexc with the construct E1 and finally E2, basing its big-step semantics on your understanding of the corresponding construct in Java.

Exercise 4  Consider the following two kinds of expressions:

(i) try (try E1 catch l1 with E2) catch l2 with E3

(ii) try E1 catch l1 with (try E2 catch l2 with E3)

Do they always evaluate to the same result? If not can you give an example of E1, E2, E3 for which different results are obtained. If so how would you go about proving it?

Exercise 5  Give a small-step semantics for the language Arithexc. Here judgements should take the form \( \langle E_1, s \rangle \rightarrow \langle E_2, s' \rangle \) and many of the defining rules should be based on the small-step semantics for arithmetic expressions given in the notes.

2  Exceptions in commands: small-step semantics

Consider the slight extension of the language While whose abstract syntax is given in Figure 5. In the assignment command \( l := E \) the expression \( E \) is allowed to come from the expression language of the previous section Arithexc and so their evaluation may raise exceptions. We also import the try ... catch l with ... construct into commands in order to allow programs to recover from exceptions.
being raised. It will also be useful to introduce the explicit command `throw E` whose only effect is to raise an exception whose value is determined by `E`. Recall that in the language `While` we also have the slightly odd command `skip` whose execution has no effect. But recall that its presence facilitated some semantic definitions, as it was taken to represent the termination of a successful execution. In the same way the command `throw E`, or rather the reduced versions `throw n` will facilitate the semantic definitions for the extended language `While^{exc}`, representing termination of an execution in which an exception has been raised.

The effect of running a command will depend on how arithmetic expressions are executed. So their semantics, both big-step and small-step will assume a semantics of arithmetic expressions such as that given in the previous section, whose judgements take the form

\[
\langle E, s \rangle \Downarrow \langle t, s' \rangle
\]

where `t` can stand for either the result of a successful evaluation, a numeral `n`, or the result of an exception being raised, an exception `throw n`.

We also require a semantics for Boolean expressions, and note that their evaluation may also raise exceptions. For example consider the result of evaluating the Boolean expression \((2 - 4) = t_1 + 2\); It should raise a `subtraction` exception. So also we assume a big-step semantics for Booleans whose judgements take the form

\[
\langle B, s \rangle \Downarrow \langle bt, s' \rangle
\]

where `bt` can either be a Boolean value `true` or `false`, the result of a successful computation, or `throw n`, an exception which has been raised.

As for the language `While`, here the judgements take the form

\[
\langle C, s \rangle \rightarrow \langle C', s' \rangle
\]

meaning: `one step in the execution of the command C relative to the state s changes the state to s' and leaves the residual command C' to be executed`. Indeed many of the defining rules for this relation over the extended language `While^{exc}` are inherited from the small-step semantics for `While`. These, called the default rules, are given in Figure 6, and most are inherited directly from Figure 3.5 of the Course Notes. The rule `(s-ass)` is slightly modified in order to reflect the more complicated semantics of arithmetics. The two new rules for the `try` command `(s-try)` and `(s-try.skip)` code up our intuitive understanding of their default behaviour.
The rules governing when exceptions are thrown are given in Figure 7. The intuition we are trying to capture is that if an exception is raised anywhere during the execution of a command it is immediately aborted and goes to an abnormal termination state, \textit{throw} \( k \) for some \( k \). This should happen for example during the execution of 

\[
\text{L} := \text{L} + 1; \text{L} := (2 - 4); \text{K} := \text{L} + 3
\]
Intuitively here the first update to the store should happen but not the third; instead the exception \( \text{throw} \, 0 \) should be raised.

These exceptions can be caught if they occur inside a \texttt{try} \ldots \texttt{catch} \, l \, \texttt{with} \, C \, ; \texttt{as with expressions the value of the exception is stored in the location} \, l \, \texttt{and the command} \, C \, \texttt{is then executed.}

The key to understanding the rules in Figure 7 is to see the connection between the commands

- \texttt{skip}: if a command \( C \) successfully terminates it will eventually be reduced to \texttt{skip}

and

- \texttt{throw n}: if a command raises an exception it will eventually be reduced to some \texttt{throw n}, representing termination because of an exception.

In the default semantics \texttt{skip} is used to manage the flow of control of commands; see for example the rules \((s\texttt{-seq.skip})\) and \((s\texttt{-try.skip})\) In a similar manner \texttt{throw n} is used to manage the flow of control when exceptions are raised. Thus in \((s\texttt{-ass.e})\) the assignment is aborted when the evaluation of the expression raises an exception. Similarly in \((s\texttt{-seq.e})\), in the execution of \( C_1 \, ; \, C_2 \) if \( C_1 \) has raised an exception,
that is has been reduced to an abnormal termination command throw \( n \), then the execution of \( C_2 \) is aborted.

The commands if \( B \) then \( C_1 \) else \( C_2 \) and while \( B \) do \( C \) throw exceptions if one is thrown in their guard \( B \); this is the import of the rules \((s\text{-con},e)\) and \((s\text{-while},e)\). Note that we do not have to worry about when exceptions are thrown in their bodies \( C_1, C_2 \) or \( C \). This will be taken care of when, if ever, they are executed.

Finally the rule \((s\text{-try},\text{skip})\) implements the expected flow of control for the try command. When running try \( C_1 \) catch \( l \) with \( C_2 \) if \( C_1 \) raises an exception it will be reduced to some throw \( n \), at which point rule \((s\text{-try},\text{skip})\) will lead to the exception being caught in location \( l \) and the execution of \( C_2 \) initiated.

**Exercise 6** For each of the following commands use the small-step semantics to determine what final store is reached, starting from the standard store \( s \).

\[
\begin{align*}
(i) & \quad (l := l + 1; l := (2 - 4)); k := l + 3 \\
(ii) & \quad (l := l + 1; l := (2 - 4)); \text{try } k := 1 \text{ catch } k \text{ with } l + 3 \\
(iii) & \quad \text{try } (\text{try } l_2 := (l_8 \div l_3) \text{ catch } l_6 \text{ with } l_4 := l_1 + 1; l_2 := l_6 - l_3) \text{ catch } l_4 \\
& \quad \text{with } (l_2 := l_4 + l_6 + l_1) \\
(iv) & \quad \text{try } l_2 := (l_2 + 1); l_1 := (l_8 \div l_3) \text{ catch } l_4 \text{ with } (l_1 := (l_6 - l_3) \text{ catch } l_4 \text{ with } l_1 := (l_4 + l_6 + l_2))
\end{align*}
\]

**Exercise 7** Consider the following statement:

\[
\langle (C_1 ; C_2); s \rangle \rightarrow^* \langle t, s' \rangle \text{ if and only if } \langle (C_1 ; (C_2 ; C_3)); s \rangle \rightarrow^* \langle t, s' \rangle
\]

where \( t \) ranges over the successful terminated command \( \text{skip} \) and the abnormal terminated commands \( \text{throw } n \).

Is this true for the language While\text{exc}? If so how would you go about proving it? If not can you give a counter-example?

**Exercise 8** Give a big-step semantics for the language of commands in Figure 5.