1 Exceptions

We included error into MinHs to be able to deal with partial functions without losing type safety, in particular progress. However, if one subexpression of a computation evaluates to error, the whole computation always evaluates to error. This is too coarse grained: in a real program, we usually only want to abort a certain subcomputation in case an error occurs and handle the error, but not abort the whole program. Most languages therefore provide some mechanism for exception handling to respond to a run-time error in a more controlled fashion. We add two new language constructs to MinHs:

- Catching an exception:
  
  \[
  \text{try } e_1 \text{ catch } x = \rightarrow e_2
  \]

  This causes \( e_1 \) to be evaluated. If a runtime error occurs in \( e_1 \), it does not cause the whole expression to evaluate to error, but instead causes the error handler \( e_2 \) to be evaluated. For the error handler, it is often important to know what kind of error occurred and what the cause was. An exception therefore returns also an error value, which is bound to \( x \) in \( e_2 \). This might be a string (error message) or integer (error code) or a more sophisticated type (in Java, for example, any object of the class Throwable).

- Raising an exception:

  \[
  \text{raise}: \exists e
  \]

  This causes the current computation to be aborted, \( e \) to be evaluated and the result directly “thrown” back to the nearest catch-expression. The raise expression is in most cases used in combination with an if-expression testing for the error condition.

In the following example, we use Haskell-like notation instead of the more restricted MinHs syntax:

\[
\begin{align*}
\text{try } ("The result is " \& show (div k l)) \\
\text{catch err } => \\
"\text{Error: } " \& err
\end{align*}
\]

\[
\begin{align*}
\text{div } x \ y = \\
\text{if } (y == 0) \text{ then } \\
\text{raise}: \text{String }"\text{Division by zero}"
\text{else } x / y
\end{align*}
\]
Here, the result of the try-expression is either a string containing the result of the division, or an error message. If the try/catch expression is part of a more complex computation, the run-time error is not visible to the outside. It is important, therefore, that both subexpressions of the try/catch-expression have the same type in a strongly typed language like Haskell or MinHs.

### 1.1 Static Semantics

We have to add the typing rules for the try/catch as well as the raise construct. Note that, as MinHs is explicitly typed. The type $\tau_{exc}$ stands for the type of the values which are thrown in case of an exception — this could be either integer values (exception code), a string (error message), or a more sophisticated type. In Java, for example, this includes all members of the type class Throwable.

\[
\frac{\Gamma \vdash e : \tau_{exc}}{\Gamma \vdash \text{raise}(\tau, e) : \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \cup \{x : \tau_{exc}\} \vdash e_2 : \tau}{\Gamma \vdash \text{try}(e_1, x.e_2) : \tau}
\]

The second rule makes sure that both branches of the try-expression do have the same type.

### 1.2 Dynamic Semantics

We will discuss two possible ways to define the dynamic semantics of exceptions here.

Once an exception is raised, the machine behaves differently: instead of simply returning the thrown value, it has to unwind the control stack until it finds the first try-frame. We model this by introducing a new machine state:

\[ s \leftarrow \text{raise}(v) \]

Now we can add the new rules for try and raise. Note that all the rules are still axioms, so we have no premises:

1. To evaluate a try-expression, evaluate the first subexpression and push the try-frame on the stack:

\[ s \triangleright \text{try}(e_1, x.e_2) \triangleright C \text{try}(\emptyset, x.e_2) \triangleright s \triangleright e_1 \]

2. If it has been successfully evaluated (exception has not been raised), pop try-frame from the stack and continue returning value:

\[ \text{try}(\emptyset, x.e_2) \triangleright s \leftarrow v_1 \rightarrow_C s \leftarrow v_1 \]

3. If exception is raised, first evaluate expression and push raise-frame on the stack

\[ s \triangleright \text{raise}(\tau, e) \rightarrow_C \text{raise}(\tau, \emptyset) \triangleright s \triangleright e \]

4. Once exception is evaluated, machine goes into exception state, pops raise-frame from stack:

\[ \text{raise}(\tau, \emptyset) \triangleright s \leftarrow v \rightarrow_C s \leftarrow \text{raise}(v) \]

5. If in exception state, and try-frame is at the top of the stack, evaluate exception handler $e_2$ and bind variable to exception value. Switch back to regular evaluation mode:

\[ \text{try}(\emptyset, x.e_2) \triangleright s \leftarrow \text{raise}(v) \rightarrow_C s \triangleright \{v/x\} e_2 \]
6. If in exception state, and top of control stack is any other frame, pop frame from stack and continue unwinding:

\[ f \triangleright s \xleftarrow{c} \text{raise}(v) \]

Alternatively, instead of rewinding the stack until we find the first try-frame, we could directly jump to the handler. This can be achieved by extending our abstract machine with a second stack which only contains the handler-frames, bundled with a snapshot of the control stack. We define a handler frame to have the form \( \text{catch}(k, x,e) \) where \( k \) is a regular control stack and \( x.e \) a handler expression. The rules for the new machine, then, are as follows:

\[
\begin{align*}
(h, k) \triangleright \text{try}(e_1, x.e_2) & \quad \mapsto_c (\text{catch}(k, x.e_2) \triangleright h, \text{try}(\emptyset) \triangleright k) \triangleright e_1 \\
(\text{catch}(k, x.e_2) \triangleright h, \text{try}(\emptyset) \triangleright k) \xleftarrow{v_1} & \quad \mapsto_c (h, k) \xleftarrow{v_1} \\
(h, k) \triangleright \text{raise}(\tau, e) & \quad \mapsto_c (h, \text{raise}(\tau, \emptyset) \triangleright k) \triangleright e \\
(h, \text{raise}(\tau, \emptyset) \triangleright k) \xleftarrow{v} & \quad \mapsto_c (h, k) \xleftarrow{\text{raise}(v)} \\
(\text{catch}(k', x.e_2) \triangleright h, k) \xleftarrow{\text{raise}(v)} & \quad \mapsto_c (h, k') \triangleright \{v/x\}e_2
\end{align*}
\]

All other rules stay the same apart from the fact that the handler stack \( h \) has to be carried along.

2 Continuations

In the previous section, we made a snapshot of the control stack to be able to directly return to a previous execution state in case of an exception. Languages like Scheme or Standard ML of New Jersey push this approach further and allow the user to make snapshots of the current control stack any time during the evaluation and restore it at a later point in time — such a snapshot is called a continuation. Continuations can be used to implement, for example, multi-threading, backtracking, and exception handling. C offers operations which serve a similar goal, sigsetjmp and longjmp.

2.1 Syntax

We add two language constructs to MinHs. First we need a means to make a snapshot of the current stack and store the result. This is done by evaluating \textbf{letcc}:

\[
\text{letcc}:: \tau \text{ in } e \text{ end}
\]

binds the current stack to \( x \) and evaluates the expression \( e \). The variable \( x \) may occur freely anywhere in \( e \). The type of \( e \) has to be given explicitly by \( \tau \). The abstract syntax representation of the above MinHs term is \text{letcc}(\tau, x,e).

Furthermore, we add a construct to restore a control stack which has been saved earlier on during the evaluation:

\[
\text{throw}:: \tau \text{ in } e_2 \text{ end}
\]

Both subexpressions are evaluated, where \( e_2 \) has to evaluate to a control stack, and \( e_1 \) to a value which is accepted by the stack of \( e_2 \) as next result. The abstract syntax representation is \text{throw}(\tau, e_1,e_2).

2.1.1 Examples

Let us first look at some examples of how \textbf{letcc} and \textbf{throw} can be used. In all three examples, the call stack has the form \textbf{plus}(1, \emptyset) \triangleright o when \textbf{letcc} is called, and is bound to variable \( x \).

- \(1 + \text{letcc } x \text{ in } 2 + (\text{throw } 3 \text{ to } x) \xmapsto{c} 4 \)

In this example, the value 3 is thrown to the control stack \( x \), which means the result is \textbf{plus}(3,1), that is 4.
• $1 + \text{letcc } x \text{ in } 2 \xrightarrow{\text{c}} 3$ Here, we have no \textit{throw} expression, so the stack is never restored, and the expression is equivalent to $1 + 2$.

• $1 + \text{letcc } x \text{ in } \text{if (throw } 7 \text{ to } x \text{) then } 3 \text{ else } 4 \xrightarrow{\text{c}} 8$ Here we can see that a \textit{throw}-expression may occur anywhere (in this case the condition of an \textit{if}-expression) since it never returns from its evaluation.

The above examples are just to demonstrate the effect of \textit{letcc} and \textit{throw} expressions. Let us now look at an example where the use of continuations does make more sense.

Consider the following problem: we want to compute the product of all elements of a list (we use Haskell, not MinHs, extended by continuation support, for this example), which is done by the following function:

```
mult [] = 1
mult (x:xs) = x * (mult xs)
```

If there is a single 0 in the list, it is already clear that the overall result of the function has to be 0. We could therefore write a slightly more efficient version which treats this case separately:

```
mult [] = 1
mult (x:xs) =
    if x == 0
        then 0
        else x * (mult xs)
```

Now, at least the computation does not traverse the rest list anymore. However, let’s say the the tenth element of the list is 0. When we reach this element, we already have nine control frames on the stack which contain suspended multiplications. So the stack still has to be unwound before we finally return 0. The next definition allows the execution to directly restore the control stack $k$ (stack at the time \textit{mult} was called) and throw the result 0 to it:

```
mult xs =
    letcc k
    in let
        mult' [] = 1
        mult' (x:xs) =
            in if (x == 0)
                then throw 0 to k
                else x * (mult' xs)
        in if (x == 0)
            then throw 0 to k
            else x * (mult' xs)
    in letcc ret in mult' xs ret
```

Continuations are first class citizens, so they can be passed to functions as arguments, or returned as result. The following definition is therefore equivalent to the previous one:

```
mult xs =
    let
        mult' [] = 1
        mult' (x:xs) k =
            in if (x == 0)
                then throw 0 to k
                else x * (mult' xs)
        in letcc ret in mult' xs ret
    in
```

Note that since they are first class citizens, stacks can be thrown as well, so expressions of the following form are possible:

```
letcc r in .... letcc s in ..... throw s r end end
```

In contrast to exceptions, it is therefore not only possible to jump back to a previous control stack with is essentially a sub stack of the current one, but to any at all. This makes an efficient implementation of continuations much harder than for exceptions.
2.2 Static Semantics

We define a stack $s$ to have type $\tau$ if its top-most frame accepts a value of type $\tau$. A continuation $\text{cont}(s)$ has type $\text{Cont}\ \tau$ if the stack $s$ has type $\tau$:

$$
\begin{align*}
\text{Stack} & : \tau \\
\text{cont}(s) & : \text{Cont}\ \tau
\end{align*}
$$

During the evaluation of an expression $e$ of type $\tau$, the current control stack has to be of type $\tau$ as well, since it is waiting to accept the result of $e$:

$$
\begin{align*}
\Gamma, x : \text{Cont} \vdash e : \tau \\
\Gamma \vdash \text{letcc}(\tau, x, e) : \tau
\end{align*}
$$

In the typing rule for $\text{throw}$, we make sure that the value which is thrown has the type expected by the continuation ($\tau_1$).

$$
\begin{align*}
\Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash e_2 : \text{Cont}\ \tau_1
\end{align*}
\Rightarrow
\begin{align*}
\text{throw}(\tau, e_1, e_2) : \tau
\end{align*}
$$

Note that the rule lets us choose any type $\tau$ for the whole $\text{throw}$ expression ($\tau$ does not occur in any of the premises of the rule).

2.3 Dynamic Semantics

The dynamic semantics of continuations is fairly straightforward, that is, the evaluation rules look simple on paper. However, as we will see in later examples, it can be quite tricky to understand the behaviour of a continuation style program.

- If the abstract machine encounters a $\text{letcc}$-expression, it binds the variable to a continuation of the current stack and evaluates its body expression:

$$
\begin{align*}
\text{letcc}(\tau, x, e) & \Rightarrow_c s \Rightarrow \{\text{cont}(s)/x\} e
\end{align*}
$$

- The evaluation of the subexpressions of $\text{throw}$ works in the same way as most other language constructs:

$$
\begin{align*}
\text{throw}(\tau, e_1, e_2) & \Rightarrow_c \text{throw}(\tau, e_1, e_2) \Rightarrow e_1 \\
\text{throw}(\tau, e_1, e_2) & \Rightarrow_c \text{throw}(\tau, e_1) \Rightarrow e_2
\end{align*}
$$

- It becomes interesting once both subexpressions are fully evaluated. According to the typing rule, the second subexpression has to be a continuation. So, the stack of the continuation is restored, and the result value of the first subexpression is returned to this stack:

$$
\begin{align*}
\text{throw}(\tau, v_1, e_2) & \Rightarrow_c v_1 < s \Rightarrow \text{cont}(s_2) \\
\Rightarrow_c s_2 < v_1
\end{align*}
$$

- The last rule just expresses the fact that a term of the form $\text{cont}(s)$ is already fully evaluated:

$$
\begin{align*}
\text{cont}(s') & \Rightarrow_c s < \text{cont}(s')
\end{align*}
$$

As an exercise, take the first three continuation examples and evaluate them using the rules given here.

For the sake of readability, we use concrete instead of abstract syntax here, and omit trivial steps not related to the evaluation of continuations:

$$
\begin{align*}
\circ \Rightarrow 1 + \text{letcc}\ x\ \text{in}\ 2 + (\text{throw}
\ 3\ \text{to}\ x)\ \text{end} \\
\Rightarrow_c (1 + \square) \Rightarrow \circ \Rightarrow \text{letcc}\ x\ \text{in}\ 2 + (\text{throw}
\ 3\ \text{to}\ x)\ \text{end} \\
\Rightarrow_c (1 + \square) \Rightarrow \circ \Rightarrow 2 + (\text{throw}
\ 3\ \text{to}\ \text{cont}(\square))\ \text{end}
\end{align*}
$$
2.4 Example

Let us now look at a more complex example: we want to write a function `compose`, which takes a continuation `cont(k)` and a function `f` as input. It returns a new continuation `cont(k1)` which, given a value `v :: a`, applies the function `f` to `v` and throws the result to `k`. The function, therefore, has the have the following type:

```
compose: (a -> b) -> Cont b -> Cont a
```

To behave in the way described, the stack `k1` has to have the following form:

```
k1 = apply(f, □) ▽ throw(□, k) ▽ k'
```

where `k'` can be any stack, since the `throw`-frame makes sure it won't be executed. The evaluation of the expression

```
throw(f(letcc(k1,??), k)
```

will lead to a stack of the required form, and bind it to `k1`. But how can we return `k1` as a result of a function? The solution is similar to the multiplication example at the start of this section: we have to make a snapshot of the stack at the call time of the function, and then throw the result (the continuation `k1` back to this stack:

```
letfun compose (a -> b) -> (Cont b) -> (Cont a) f k =
  letcc r
  in
  throw (f (letcc k1 in throw k1 to r end)) to k
```

2.5 Exceptions and Continuations

So, how can we use continuations to implement the exceptions discussed in the previous section? We have to translate the two language constructs:

- `try(e1, x; e2)` and
- `raise(v)`

into `letcc`'s and `throw`'s. So, let's think about what effects these two language constructs have. Calling `try(e1, x; e2)` does two things: firstly, it makes a snapshot of the current control stack and continues with the evaluation of `e1`; secondly, it makes sure that if the exception is thrown, the exception value will be bound to `x` in `e2`. At this point in the program we can only model the first step using continuations (we have to take care of the second part when the exception is raised):

```
• letcc(k, e1)
```

The control stack is now bound to `k`. Now, when the exception is raised, we have to throw the exception value to the snapshot of the stack we saved (`k`) and make sure it is bound to `x` in `e2`. This can be achieved by pushing the function `fun(f.x.e2)` on the stack `k` using the function `compose`.

```
• throw(v, compose(fun(f.x.e2), k))
```