1 Exceptions

Recall by-value PCF with computation type. The essence is to make lazy evaluation explicit on
the static level via modal separation between "value" and "computation".

The key rules that handle the modality are presented below:

\[\Gamma \vdash e \sim \tau\]
\[\Gamma \vdash \text{comp}(e) : \tau\]
\[\Gamma \vdash e_1 : \text{comp}(\tau_1) \quad \Gamma, x : \tau \vdash e_2 \sim \tau_2\]
\[\Gamma \vdash \text{let}\{e_1; x. e_2\} \sim \tau_2\]

where the dynamics are defined using the following rules:

\[\text{comp}(e) \text{ val} \]
\[e \rightarrow e'\]
\[\text{let}\{\text{comp}(e); x. e\} \rightarrow \text{let}\{\text{comp}(e'); x. e\}\]
\[v \text{ val} \]
\[\text{let}\{\text{comp}(v); x. e\} \rightarrow [v/x]e\]

We extend the language with a with the following construct for exceptions.

\[e ::= \ldots\]
\[\text{raise}[\rho](e)\]

The statics for \text{raise} are given below:

\[\Gamma \vdash e : \tau_{\text{exn}}\]
\[\Gamma \vdash \text{raise}[\rho](e) \sim \rho\]

Note that we have not yet defined the \(\tau_{\text{exn}}\) type: the type of exception values. There are many
types that we could choose to be \(\tau_{\text{exn}}\), including:

- Strings, to contain error messages.

\(^1\)\text{bind} and \text{let} are the same thing in the context of this class.
• Ints, to contain error codes.
• A sum of many different types, to represent a variety of possible errors.

However, all of these have the shortcoming that they constrain us to storing one particular type (or a fixed number of types) of information in an exception. However, there are many different conditions under which we might want to raise an exception, and in each of them, we might want to associate a different type of data with the exception. We will see a way of dealing with this problem later in the class, so for now we won’t define exactly what $\tau_{\text{exn}}$ is.

The computation type and $\text{bnd/let}$ constructs come in handy to define the dynamics of $\text{raise}$. An expression of type $\text{comp}\{\tau\}$ now represents a suspended computation that will either evaluate to a value of type $\tau$, or trigger an exception. "Bind" should handle both of the cases where the suspended computation could end up with.

The typing rules says it all:

$$
\Gamma \vdash e : \text{comp}\{\tau\} \quad \Gamma, x_1 : \tau_1 \vdash e_1 \approx \tau_2 \quad \Gamma, x_2 : \tau_{\text{exn}} \vdash e_2 \approx \tau_2
\Rightarrow
\Gamma \vdash \text{bnd}\{e; x_1.e_1, x_2.e_2\} \approx \tau_2
$$

2 Control Flow

So far in this course, we have been using structural dynamics to study the evaluation of terms in a language. After an expression is deemed well-formed by the statics, the transition rules in a dynamics system tell us what happens at every step of evaluation, and the valuation rules tell us when evaluation ceases.

We’ve been able to encode some notion of control flow into this system: if-then-else clauses, function calls and recursive functions, as well as laziness in evaluation. But all of these are local forms of control, solely determined by the two sides of the $\mapsto$ judgment. Advanced languages have sophisticated methods of nonlocal control flow, among which are the ideas of exceptional control flow, continuations, nonlocal jumps ($\text{setjmp}$, $\text{longjmp}$), etc. Such constructs must be modeled with a more sophisticated consideration of control.

The first half of this course largely focuses on logical fundamentals, safety, semantics, and expressiveness of the languages we develop. In the second half, we begin to examine interesting characteristics of languages like scope, mutable assignable, and call stacks. Control stacks in particular allow us to model nonlocal control, and the logical system that corresponds with our understanding of call stacks is called a $K$ machine.

2.1 K Machines with exceptions

$K$ machine is an abstract machine that lets us model computation. (A Turing machine is also an abstract machine of sorts.) It’s not a language with syntax like we have studied so far but it allows as to specify language semantics in a different way.

We define the stacks $k$:

$$
k := \epsilon \mid k; f
$$

As you can see, stacks themselves are just lists of stack frames. $\epsilon$ is the empty stack.

---

2"stack machine" in the lecture
The semantics of K machines are defined by the following judgments:

\( k \triangleright e \) means that we are evaluating computation \( e \) on the stack \( k \)

\( k \triangleleft v \) means that we are returning the value \( v \) from the stack \( k \)

\( k \blacktriangleleft e \) means that we are passing the exception value \( e \) to stack \( k \)

There is also a transition judgment \( \rightarrow \), which relates the previous two judgments. The key rules are the following:

\[
\frac{k \triangleright \text{ret}(v) \rightarrow k \triangleleft v}{k; \triangleright \text{raise}(e) \rightarrow k \blacktriangleleft e}
\]

\[
k \triangleright \text{bnd}\{\text{comp}(e); x_1.e_1, x_2.e_2\} \rightarrow k; \begin{cases} x_1.e_1 \\ x_2.e_2 \end{cases} \triangleright e
\]

\[
k; \begin{cases} x_1.e_1 \\ x_2.e_2 \end{cases} \triangleleft v \rightarrow k \triangleright [v/x_1]e_1
\]

\[
k; \begin{cases} x_1.e_1 \\ x_2.e_2 \end{cases} \blacktriangleleft v \rightarrow k \triangleright [v/x_2]e_2
\]

\[
f \not= \begin{cases} x_1.e_1 \\ x_2.e_2 \end{cases}
\]

\[
k; f \blacktriangleleft v \rightarrow k \blacktriangleleft v
\]

A valid K machine is an explicit representation of the control flow of a PCF program. As an exercise, try to evaluate \( (\text{fun}\{f.x.\text{ret}(s(x))\})(z) \) on a K machine.

### 2.1.1 Safety

K machines have a notion of correctness which should be shown to make the system fully logically sound and complete. In order to formalize this notion, we define a judgement \( K \perp \tau \) that says the stack \( K \) "accepts" an value of type \( \tau \).

Empty stack accepts any type:

\[
\epsilon \perp \tau
\]

Each frame should "match" with the stack preceeding it:

\[
\frac{K \perp \tau \quad x : \tau' \vdash e \perp \tau}{K; x.e \perp \tau'}
\]

\[
\frac{K \perp \tau \quad x_1 : \tau' \vdash e_1 \perp \tau \quad x_2 : \tau_{\text{exn}} \vdash e_2 \perp \tau}{K; \begin{cases} x_1.e_1 \\ x_2.e_2 \end{cases} \perp \tau'}
\]

Then we can define a new judgement \( \text{ok} \) that says a state of a K machine is well-formed:

\[
\frac{e \perp \tau}{K \triangleright e \text{ ok}}
\]

\[
\frac{v : \tau}{K \triangleleft v \text{ ok}}
\]

\[
\frac{e \perp \tau}{K \triangleright e \text{ ok}}
\]
3 Exceptions vs. Errors

Our treatment of exceptions is modeled off the behavior of exceptions in ML. Exceptions are one of the frequently controversial aspects of high-level programming languages, and much debate occurs over whether they should be used, what information they should carry, how they should be checked statically, the importance of optimizing exceptional control flow, and even whether they’re necessary at all!

In a functional setting, exceptional control flow circumvents the type system. If a type is a theorem and a function is an implication according to Curry-Howard, then the presence of exceptions undermines the notion of proof since \texttt{raise} can be used to prove any claim. It also makes evaluation “dangerous” in that well-typed expressions may now not only evaluate to a value, or diverge, but also now raise an exception. Why, then, do we still use exceptions?

It turns out that ML’s exception features are highly nuanced largely thanks to the \texttt{exn} type.

Exceptions in ML serve many more purposes than meets the eye:

- To indicate that an error has occurred. A generic “failure” would also suffice to show this, as we saw with dynamically checked languages, or perhaps a sum (option) type.
- To deliberately invoke nonlocal control flow, such as in a backtracking algorithm. This pattern is much more common in ML than in e.g. Java. Regular control flow could of course be written to achieve the same effect, but might be convoluted by comparison. Continuations, which we will see soon, also serve this purpose.
- To share data with the handler of the exception according to a set of programmer-defined tags.

The last point distinguishes ML-style exceptions from other languages, and merits some more discussion. For now we will appeal to our understanding of the \texttt{exn} type from ML to see how an exception is able to transmit a packet of data to a handler, who can only unwrap the data if they understand the exception. This is a rudimentary form of secrecy in data transmission!

Consider the following snippet:

\begin{itemize}
  \item To indicate that an error has occurred. A generic “failure” would also suffice to show this, as we saw with dynamically checked languages, or perhaps a sum (option) type.
  \item To deliberately invoke nonlocal control flow, such as in a backtracking algorithm. This pattern is much more common in ML than in e.g. Java. Regular control flow could of course be written to achieve the same effect, but might be convoluted by comparison. Continuations, which we will see soon, also serve this purpose.
  \item To share data with the handler of the exception according to a set of programmer-defined tags.
\end{itemize}
structure Alice :> sig
  exception Message of string
  val tell : string -> unit
end =
struct
  exception Message of string
  fun tell s = raise Message "secret"
end

Now we may communicate with Alice by sending using `tell` and receiving by handling `Alice.Message`:

fun tell_and_listen s = (Alice.tell s; NONE) handle Alice.Message s => SOME s
val SOME "secret" = tell_and_listen "hi"

Now consider sealing the exception away:

signature SHH = sig
  val tell : string -> unit
end
structure SecretAlice = Alice :> SHH

Now we can still send messages with `tell`, but there is no way to receive a message, because the exception is completely opaque to the sender. A handler could try to catch the exception, but has no way to unwrap the internals since there is no exception `Message` visible to the outside. Only someone who had access to the exception `Message` could read Alice’s message, in a typesafe system!

We will discuss `exn` in much more detail when we discuss dynamic classification later in the course. For now, this is a taste of how interesting ML exceptions really are, with a control component that is already more ergonomic than other languages, and a data component that is innovative. Despite some drawbacks, using exceptions judiciously enables concise and nuanced code. Few other inventions could suffice to serve all the purposes of exceptions.