1 Accounting For Errors

In lecture this week, we saw a number of extensions to MinML, including sums and pairs. Here, we will explore in more detail another extension from lecture: run-time errors. To do so we will add another primitive operator over integers, division. Unlike addition, subtraction, and multiplication, the division of integers is a partial function. That is, it does not yield a result for all possible inputs. In particular, consider the expression \( \text{div}(\text{num}(2), \text{num}(0)) \). We would like to include division in our type-safe language, but so far we have no way of accounting for what “happens” when we evaluate a division by zero.

(One possibility is to add an additional value of type \text{int} that is the result of such an expression. This value is sometimes called “NaN” or “not-a-number” when it appears in specifications of floating-point arithmetic. If we were to do so, however, we would have other problems to consider; for example, what is the result of \text{num}(1) = \text{NaN}?)

1.1 Stuck Made Explicit

We will add a new expression to our language, shown below, to capture this state. (This expression is also sometimes known \text{wrong} or as the “stuck state.”)

\[
e ::= \ldots \mid \text{error}
\]

(\text{error} a value? Why or why not? It may become more clear when we introduce a typing rule for \text{error} below.)

With \text{error} in hand, we can give an evaluation rule that applies to the expression above.

\[
\text{div}(\text{num}(k), \text{num}(0)) \mapsto \text{error} \quad \text{DivZero}
\]

We haven’t quite finished with evaluation yet, however: consider the following expression:

\[
\text{if}(\text{div}(\text{num}(2), \text{num}(0)), \ldots) \mapsto \text{if}(\text{error}, \ldots) \mapsto ?
\]
Even though we’ve made progress with division, we still are stuck at the if. We will need to add new rules to propagate errors through all of our existing constructs. Analogously to our search evaluation rules, we add:

\[
\begin{align*}
\text{apply(error}, e_2) & \mapsto \text{error} \\
\text{if(error}, e_1, e_2) & \mapsto \text{error} \\
\text{let(error}, x.e) & \mapsto \text{error} \\
\text{add(v_1, \ldots, v_{j-1}, error}, e_{j+1}, \ldots) & \mapsto \text{error}
\end{align*}
\]

Similarly for the other primitive operations.

### 1.2 Typing For Errors

Before we can go ahead and extend our safety proof, we must give a type to our new expression. Since no actual computation is performed once we have encountered an error, we can assign any type to an expression that has failed (i.e. there is no way to distinguish one error from another).

\[\Gamma \vdash \text{error} : \tau\]  \hspace{1cm} ErrorTyp

**Preservation**

If \( \cdot \vdash e : \tau \) and \( e \mapsto e' \), then \( \cdot \vdash e' : \tau \). We have previously shown this proof by induction over the derivation of \( e \mapsto e' \), so we have six new cases to consider. We show only two.

**Rule DivZero** \( e = \text{error} \)

There are no assumptions to this rule, so we have no subderivations to consider. However, we only need to show that \( e' : \tau \). Since \( e = \text{error} \), this is easy enough.

\( \text{error} : \tau \) \hspace{1cm} By rule

**Rule IfError** \( e = \text{error} \)

Again we have no assumptions and so, again, no subderivations. In fact, this case looks just like the last case!

\( \text{error} : \tau \) \hspace{1cm} By rule

All of our new cases for preservation look exactly like this since each evaluates (in one step) to the error expression. With these new cases, our extended proof of preservation is complete.
Progress

Here we must extend the theorem: if $\cdot \vdash e : \tau$ then either

i. $e$ value or

ii. $e \mapsto e'$ for some $e'$ or

iii. $e$ is error

This proof was given by rule induction over the derivation of $\cdot \vdash e : \tau$, and we have one new typing rule to consider, so we have one additional case.

Rule ErrorTyp $e = \text{error}$

$e$ is error \hspace{1cm} \text{By assumption}

Easy enough! Have we finished? No, because we have extended the induction hypothesis, we have an additional subcase to consider each time we applied it.

Consider the case for IfTyp:

\[
\begin{array}{c}
\vdash e : \text{bool} \quad \vdash e_1 : \tau \quad \vdash e_2 : \tau \\
\hline
\vdash \text{if}(e, e_1, e_2) : \tau
\end{array}
\]

Previously, we applied the induction hypothesis to the first subderivation to conclude:

Either $e$ value or $e \mapsto e'$

Now must must consider each of:

 Either $e$ value or $e \mapsto e'$ or $e$ is error

The first two subcases are identical to those in our old proof, but we must finish the third.

\[
\begin{array}{c}
e \text{ is error} \hspace{1cm} \text{By case (iii) of i.h.} \\
\text{if(error, } e_1, e_2) \mapsto \text{error} \hspace{1cm} \text{By rule}
\end{array}
\]

We have shown that there is a step to be made and so progress is maintained.

In each of the applications of the induction hypothesis, we will have a new subcase, and (if we’ve set things up correctly) we should have a new rule to apply. If we find a subcase and no rule to apply, it probably means that we’ve forgotten a rule; conversely, if a new rule doesn’t apply anywhere, it was probably unnecessary.

(Is it clear now why we don’t want \text{error} to be a value? Think about value inversion with respect to error.)
1.3 Coming Attractions

You may feel as though you’ve missed half the fun: we discussed how to raise errors, but we haven’t given any hint of how handle them. As you might have guessed from the language we’ve used to talk about errors, we will soon be covering exactly these ideas in an upcoming lecture on exceptions.