# COMP 321 Homework 5, Fall 2021: Exceptions via Sum Types

In the previous homework, you got some practice using exceptions. In this homework, you will complete the translation of exceptions to sum types sketched in Lecture 17. This is an example of a *compiler pass* or *language to language translation*. This means there will be two programming languages in play in this assignment:

- The *source language* is the input to the translation. This will be a variant PCF with exceptions.
- The *target language* is the output of the translation. This will be a variant of PCF with product and sum types — as in the midterm or the previous homework. If you like, you can assume the target language has nested pattern matching, as you implemented in Homework 4.

## 1 Source Language

The source language contexts, types, and expressions are defined as follows:

$$\begin{array}{lcl} \Gamma & ::= & \cdot \mid \Gamma, x : \tau \mid \Gamma, x :^{\mathsf{n}} \tau \\ \tau & ::= & \mathsf{nat} \mid \tau_1 \to \tau_2 \\ e & ::= & x \mid u \mid \mathsf{zero} \mid \mathsf{succ}(e) \mid \mathsf{case}\{e, e_0, x.e_1\} \mid \lambda x : \tau.e \mid e_1\, e_2 \mid \mathsf{fix}\, x : \tau.e \mid \mathsf{raise}(e) \mid e_1\, \mathsf{handle}\, x \Rightarrow e_2 \end{array}$$

The source language has the following typing rules:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{nat}} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(e) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathsf{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathsf{case}\{e, e_0, x.e_1\} : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau}$$

$$\frac{u :^{\mathsf{n}} \tau \in \Gamma}{\Gamma \vdash u : \tau} \qquad \frac{\Gamma, u :^{\mathsf{n}} \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, u : \tau.e : \tau}$$

$$\frac{\Gamma \vdash e : \mathsf{exn}}{\Gamma \vdash \mathsf{raise}(e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \mathsf{exn} \vdash e_2 : \tau}{\Gamma \vdash e_1\, \mathsf{handle}\, x \Rightarrow e_2 : \tau}$$

In the context $\Gamma$, we distinguish between regular variables $x : \tau$ and *by-name* variables $u :^{\mathsf{n}} \tau$. In a call-by-value operational semantics, variables almost always are substituted by values. However, $\mathsf{fix}\, u : \tau.e$ substitutes a non-value expression for the variable $u$ when it unrolls the recursion. To define the translation from exceptions to sum types, we will need to distinguish the lambda- or case-bound variables (which stand for values) from the fix-bound variables (which stand for general expressions).

In these rules, the type $\mathsf{exn}$ can be chosen to be any type (e.g. a recursive sum for all exception constructors in the program). For this assignment, for simplicity, we define $\mathsf{exn}$ to be $\mathsf{nat}$, i.e. we number all of the expcetions in the program (like $\mathsf{TypeError}$ is exception 1, $\mathsf{DivideByZero}$ is exception 2, $\ldots$) and don't account for exceptions that carry data.

The operational semantics for exceptions consists of three types of rules: instruction rules and search rules as usual, along with *propogation rules* that propogate a raised exception up to the nearest enclosing handler, or to the top of the expression.

$$\frac{}{\textsf{zero}\ \textsf{value}} \qquad \frac{e\ \textsf{value}}{\textsf{succ}(e)\ \textsf{value}} \qquad \frac{}{\lambda x : \tau . e\ \textsf{value}}$$

$$\frac{e\ \textsf{value}}{e\ \textsf{done}} \qquad \frac{e\ \textsf{value}}{\textsf{raise}(e)\ \textsf{done}}$$

$$\frac{e \mapsto e'}{\textsf{succ}(e) \mapsto \textsf{succ}(e')}\ \texttt{suc-search} \qquad \frac{\textsf{raise}(e)\ \textsf{done}}{\textsf{succ}(\textsf{raise}(e)) \mapsto \textsf{raise}(e)}\ \texttt{suc-prop}$$

$$\frac{e \mapsto e'}{\textsf{case}\{e, e_0, x.e_1\} \mapsto \textsf{case}\{e', e_0, x.e_1\}}\ \texttt{case-search} \qquad \frac{\textsf{raise}(e)\ \textsf{done}}{\textsf{case}\{\textsf{raise}(e), e_0, x.e_1\} \mapsto \textsf{raise}(e)}\ \texttt{case-prop}$$

$$\frac{}{\textsf{case}\{\textsf{zero}, e_0, x.e_1\} \mapsto e_0}\ \texttt{case-instr-0} \qquad \frac{e\ \textsf{value}}{\textsf{case}\{\textsf{succ}(e), e_0, x.e_1\} \mapsto e_1[e/x]}\ \texttt{case-instr-1}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \texttt{app-search-1} \qquad \frac{e_1\ \textsf{value} \quad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \texttt{app-search-2}$$

$$\frac{(\textsf{raise}(e))\ \textsf{done}}{(\textsf{raise}(e))\ e_2 \mapsto \textsf{raise}(e)}\ \texttt{app-prop-1} \qquad \frac{e_1\ \textsf{done} \quad \textsf{raise}(e)\ \textsf{done}}{e_1\ (\textsf{raise}(e)) \mapsto \textsf{raise}(e)}\ \texttt{app-prop-2}$$

$$\frac{e_2\ \textsf{value}}{(\lambda x : \tau . e)\ e_2 \mapsto e[e_2/x]}\ \texttt{app-instr}$$

$$\frac{}{\textsf{fix}\, u : \tau . e \mapsto e[(\textsf{fix}\, u : \tau . e)/u]}\ \texttt{fix-instr}$$

$$\frac{e \mapsto e'}{\textsf{raise}(e) \mapsto \textsf{raise}(e')}\ \texttt{raise-search} \qquad \frac{\textsf{raise}(e)\ \textsf{done}}{\textsf{raise}(\textsf{raise}(e)) \mapsto \textsf{raise}(e)}\ \texttt{raise-prop}$$

$$\frac{e_1 \mapsto e_1'}{(e_1\ \textsf{handle}\, x \Rightarrow e_2) \mapsto (e_1'\ \textsf{handle}\, x \Rightarrow e_2)}\ \texttt{handle-search}$$

$$\frac{\textsf{raise}(e)\ \textsf{done}}{(\textsf{raise}(e)\ \textsf{handle}\, x \Rightarrow e_2) \mapsto e_2[e/x]}\ \texttt{handle-catch}$$

$$\frac{e_1\ \textsf{value}}{(e_1\ \textsf{handle}\, x \Rightarrow e_2) \mapsto e_1}\ \texttt{handle-value}$$

Note that `case-instr-1` and `app-instr` and `handle-catch` substitute values for regular variables, while `fix-instr` substitutes an expression for a by-name variable.

## 2 Target Language

The target language is call-by-value PCF/FPC (see PFPL Chapter 19 and 20) with product and sum types (Chapter 10 and 11). You can use the concrete syntax from Homework 4 to write expressions in this language, and assume that the type $\mathsf{nat}$ with $\mathsf{zero}, \mathsf{succ}(e), \mathsf{case}\{e, e_0, x.e_1.\}$ has been defined analogously to that homework.

The most important thing is that *the target language does not have exceptions*, so we will need to translate exceptions to uses of sum types.

In stating the correctness of the translation, we will use *definitional equality* of target language programs. We write $\Gamma \vdash e \equiv e' : \tau$ to mean "$e$ is equal to $e'$ modulo the operational semantics". This relation is defined for well-typed programs with the same type in the same context, $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$. Intuitively, the definitional equality is defined by

- All of the usual step rules are definitional equality rules, e.g.

$$\frac{\Gamma, x : \tau_2 \vdash e : \tau \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_2\ \mathsf{ovalue}}{\Gamma \vdash (\lambda x : \tau.e)\ e_2 \equiv e[e_2/x] : \tau}$$

  and similarly for the instruction steps for natural numbers, products, sums, recursive types, etc.

  Moreover, these step rules apply *in any context*, including when an expression has free variables.

  We write $\Gamma \vdash e\ \mathsf{ovalue}$ for an extension of the usual value predicate $e\ \mathsf{value}$ to include regular variables (which stand for values):

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x\ \mathsf{ovalue}} \qquad \frac{}{\Gamma \vdash \lambda x : \tau.e\ \mathsf{ovalue}} \qquad \frac{\Gamma \vdash e\ \mathsf{ovalue}}{\Gamma \vdash \mathsf{inleft}_\tau(e)\ \mathsf{ovalue}} \qquad \frac{\Gamma \vdash e\ \mathsf{ovalue}}{\Gamma \vdash \mathsf{inright}_\tau(e)\ \mathsf{ovalue}}$$

$$\frac{\Gamma \vdash e_1\ \mathsf{ovalue} \quad \Gamma \vdash e_2\ \mathsf{ovalue}}{\Gamma \vdash \langle e_1, e_2 \rangle\ \mathsf{ovalue}} \qquad \frac{\Gamma \vdash e\ \mathsf{ovalue}}{\Gamma \vdash \mathsf{fold}_{t.\tau}(e)\ \mathsf{ovalue}}$$

  All of the $e\ \mathsf{value}$ premises on the step rules become $\Gamma \vdash e\ \mathsf{ovalue}$ premises on the definitional equality rules. E.g. you can reduce a function application when the argument is an "open value", i.e. an expression constructed from the value constructors *and from variables*.

- Definitional equality is an equivalence relation: it is reflexive, symmetric, and transitive.

- Definitional equality is a congruence: you can replace equals with equals anywhere in a term, via search rules like

$$\frac{\Gamma, x : \tau_1 \vdash e \equiv e' : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e \equiv \lambda x : \tau_1.e' : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \equiv e_1' : \tau_2 \to \tau \quad \Gamma \vdash e_2 \equiv e_2' : \tau_2}{\Gamma \vdash e_1\ e_2 \equiv e_1'\ e_2' : \tau}$$

The upshot of all of this is that definitional equality lets you "step anywhere in an expression, as many times as you want, in any direction". If you like, you can see PFPL Section 5.4 for more details, but this informal understanding will be enough to prove the correctness of the translation from exceptions to sum types.

## 3 The Translation

In the section, you will define a language-to-language translation that compiles exceptions to sum types, as discussed at the end of Lecture 17.

The basic structure of this translation is that a source-language program

$$x_1 : \tau_1, \ldots, x_n : \tau_n, u_1 :^{\mathsf{n}} \sigma_1, \ldots, u_m :^{\mathsf{n}} \sigma_m, \vdash e : \tau$$

is translated to a target language program

$$x_1 : |\tau_1|, \ldots, x_n : |\tau_n|, u_1 :^{\mathsf{n}} |\sigma_1| + \mathsf{exn}, \ldots, u_m :^{\mathsf{n}} |\sigma_n| + \mathsf{exn}, \vdash |e| : |\tau| + \mathsf{exn}$$

The notation $|e|$ refers to a function from source-language expressions to target language-expressions, and the notation $|\tau|$ refers to a translation from source-language types to target-language types.

Intuitively, this means that a source language program of type $\tau$ becomes a target language program of type $|\tau| + \mathsf{exn}$. That is, a source language program with type $\tau$ in some sense "lies" about its type — it doesn't always return a $\tau$; sometimes it raises an exception. The translation makes this "lie" explicit using a sum type, where $|e| : |\tau| + \mathsf{exn}$ either returns a $|\tau|$ (tagged with `inleft` sum type constructor), or it returns an exception (tagged with the `inright` sum type constructor). The translation of the context says that a regular variable $x$ has type $|\tau_i|$ — because a regular variable stands for a value, which is done, the expression that is substituted for it doesn't raise an exception. On the other hand, by-name variables $u : \sigma_i$ are translated to have type $|\sigma_i| + \mathsf{exn}$, because the non-value that is substituted for them might raise an exception.

The type translation $|\tau|$ is needed because a function $\tau_1 \to \tau_2$ in the source language might sometimes raise an exception when it is called:

$$\begin{array}{rcl} |\mathsf{nat}| & = & \mathsf{nat} \\ |\tau_1 \to \tau_2| & = & |\tau_1| \to (|\tau_2| + \mathsf{exn}) \end{array}$$

Note that since we take $\mathsf{exn}$ to be $\mathsf{nat}$, if we write $\mathsf{exn}$ for $\mathsf{nat}$ in the target language, then we have $|\mathsf{exn}| = \mathsf{exn}$ as well.

To make the above easier to write, we define the translation on typing contexts by

$$\begin{array}{rcl} |\emptyset| & = & \emptyset \\ |\Gamma, x : \tau| & = & |\Gamma|, x : |\tau| \\ |\Gamma, u :^{\mathsf{n}} \sigma| & = & |\Gamma|, u :^{\mathsf{n}} |\sigma| + \mathsf{exn} \end{array}$$

and then the typing specification for the translation is

**Static Correctness:** If $\Gamma \vdash e : \tau$ then $|\Gamma| \vdash |e| : |\tau| + \mathsf{exn}$

## 3.1 Definition of the Translation and Static Correctness

Here are the cases of the translation of expressions that we did in class:

```
| zero   | = inleft zero
| succ e | = case |e| { inleft x => inleft (succ x)
                      | inright y => inright y }
| raise e | = case |e| { inleft x => inright x
                       | inright y => inright y}
| e1 handle x => e2 | =  case |e|
                           { inleft y => inleft y
                           | inright x => |e2| }
```

A model case of the proof of static correctness is:

- Case for $\mathsf{succ}(e)$. Assume we have

$$\frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}(e) : \mathsf{nat}}$$

  By the inductive hypothesis, $|\Gamma| \vdash |e| : |\mathsf{nat}| + \mathsf{exn}$, but by definition $|\mathsf{nat}| = \mathsf{nat}$. We need to show that $|\Gamma| \vdash |\mathsf{succ}(e)| : \mathsf{nat} + \mathsf{exn}$. By definition, `|succ e|` is

```
case |e| { inleft x => inleft (succ x) | inright y => inright y }
```

  and for this to have type $\mathsf{nat} + \mathsf{exn}$, it suffices to show that $|\Gamma|, x : \mathsf{nat} \vdash \mathsf{inleft}(x) : \mathsf{nat} + \mathsf{exn}$ and $|\Gamma|, y : \mathsf{exn} \vdash \mathsf{inright}(y) : \mathsf{nat} + \mathsf{exn}$, which are true by the typing rules for inleft and inright.[1]

**Task 1** (50%). Finish defining the translation:

$$\begin{array}{rcl} |x| & = & \ldots \\ |u| & = & \ldots \\ |\lambda x : \tau . e| & = & \ldots \\ |e_1\, e_2| & = & \ldots \\ |\mathsf{case}\{e, e_0, x.e_1\}| & = & \ldots \\ |\mathsf{fix}\, u : \tau . e| & = & \ldots \end{array}$$

(To define the translation, we assume that we can tell whether a variable is a regular variable $x$ or a by-name variable $u$ e.g. by the variable name). For each, give the case of the proof of static correctness for the typing rule for that term (you can either write a typing derivation or describe it as in the model case above).

## 3.2 Dynamic Correctness

While static correcness says that the target program has the intended type, *dynamic correctness* says that the operational semantics of the source program is preserved by the translation.

**Dynamic Correctness:** If $\cdot \vdash e : \tau$ and $\cdot \vdash e' : \tau$ and $e \mapsto e'$ then $\cdot \vdash |e| \equiv |e'| : |\tau| + \mathsf{exn}$.

Here are a couple of model cases of the proof:

- For `succ-search`: Assume $\cdot \vdash \mathsf{succ}(e) : \tau$ and $\cdot \vdash \mathsf{succ}(e') : \tau$ and

$$\frac{e \mapsto e'}{\mathsf{succ}(e) \mapsto \mathsf{succ}(e')}\ \texttt{suc-search}$$

  By inversion on typing, $\tau$ and $\tau'$ are $\mathsf{nat}$ and $\cdot \vdash e : \mathsf{nat}$ and $\cdot \vdash e' : \mathsf{nat}$, so the inductive hypothesis on $e \mapsto e'$ gives that $|e| \equiv |e'| : |\mathsf{nat}| + \mathsf{exn}$. By congruence of definitional equality,

```
case |e| { inleft x => inleft (succ x) | inright y => inright y }
```

  and

[1] You can leave off the type annotation on inleft/inright for this problem.

```
case |e'| { inleft x => inleft (succ x) | inright y => inright y }
```

are definitionally equal as well.

- For `succ-prop`: Assume $\cdot \vdash \mathsf{succ}(\mathsf{raise}(e)) : \mathsf{nat}$ and $\cdot \vdash \mathsf{raise}(e) : \mathsf{nat}$ and

$$\frac{\mathsf{raise}(e)\ \mathsf{done}}{\mathsf{succ}(\mathsf{raise}(e)) \mapsto \mathsf{raise}(e)}\ \texttt{suc-prop}$$

Expanding definitions, we need to show that

```
case (case |e| { inleft x => inright x
               | inright y => inright y})
     { inleft x => inleft (succ x)
     | inright y => inright y }
```

is definitionally equal to

```
(case |e| { inleft x => inright x
          | inright y => inright y})
```

Since `raise e` is done by assumption, `e` is a value. Therefore, by Lemma 1, `|e|` is definitionally equal to `inleft v` for some value `v:exn`. Thus, both expressions are definitionally equal to `inright v` by the instruction steps for case analysis, and therefore they are equal.

**Task 2** (50%). Prove the cases of dynamic correctness for the remaining step rules `case-search`, `case-prop`, `case-instr-0`, `case-instr-1`, `app-search-1`, `app-search-2`, `app-prop-1`, `app-prop-2`, `app-instr`, `fix-instr`, `raise-search`, `raise-prop`, `handle-search`, `handle-catch`, `handle-value`.

Your proof can use the following lemmas:

- Lemma 1 (Translation of a value): If $\cdot \vdash e : \tau$ and $e\,\mathsf{value}$ then $\cdot \vdash |e| \equiv \mathsf{inleft}(v) : |\tau| + \mathsf{exn}$ for some $\cdot \vdash v : |\tau|$ such that $v\,\mathsf{value}$. That is, the translation of a value is always inleft (because a value doesn't raise an exception) of a value.
- Lemma 2 (Substitution for value variable): If $x : \tau_2 \vdash e : \tau$ and $\cdot \vdash e_2 : \tau_2$ and $e_2\,\mathsf{value}$, then $\cdot \vdash |e[e_2/x]| \equiv |e|[v/x] : \tau$ for the value $v$ such that $|e_2| \equiv \mathsf{inleft}(v)$ given by Lemma 1.
- Lemma 3 (Substitution for by-name variable): If $u :^{\mathsf{n}} \tau_2 \vdash e : \tau$ and $\cdot \vdash e_2 : \tau_2$ then $\cdot \vdash |e[e_2/u]| \equiv |e|[|e_2|/u] : \tau$

You don't need to show the proofs of these lemmas, but you should make sure they are true for your translation.