Lenguajes de Programación 2017-1
Nota de clase 12: Paradigma Imperativo III / Excepciones*

Favio E. Miranda Perea Lourdes del Carmen González Huesca
Facultad de Ciencias UNAM

5 de noviembre de 2018

1. Introducción

La programación en el mundo real está llena de situaciones donde una función o procedimiento no pueda terminar su ejecución y deba reportar tal hecho a su emisor. Algunas de estas situaciones son:

- División entre cero.
- Desborde del rango aritmético.
- Índice de un arreglo fuera de rango.
- Llave de búsqueda faltante en una base de datos.
- Archivo no encontrado.
- Archivo no accesible.
- Sistema sin memoria.
- Proceso terminado por el usuario.

Algunas de estas condiciones pueden manejarse cambiando el tipo del rango de la función por un tipo suma o variante, por ejemplo la función predecesor puede devolver un valor de tipo $\mathsf{Maybe\ Nat} =_{def} 1 + \mathsf{Nat}$ en vez de Nat y devolverá el valor trivial () en caso de que su argumento sea cero. Sin embargo en muchas situaciones donde la situación a manejar es en verdad excepcional no queremos forzar a que cada llamada de la función considere la posibilidad de que este caso suceda. En su lugar preferimos que la condición excepcional cause una transferencia directa del control del programa a un manejador de excepciones definido en otro nivel del programa o, en el caso extremo, simplemente abortar totalmente el programa. En esta nota nos encargamos de estudiar formalmente el concepto de excepción con y sin manejo.

*Estas notas se basan en los libros de Harper y Pierce y en material de Gerwin Klein y Frank Pfenning.

2. Excepciones sin manejo

La manera más simple de introducir excepciones al lenguaje es mediante un constructor error que causa la detención total de un programa. Bosquejamos aquí su uso y significado al agregarlo al cálculo lambda con tipos simples $\lambda^{\rightarrow}$.

Ambas sintaxis se extienden con una constante de error:

$$e ::= \ldots \mid \mathsf{error}$$

La evaluación se extiende como sigue:

$$\frac{}{\mathsf{app}(\mathsf{error}, e_2) \rightarrow \mathsf{error}}$$

$$\frac{}{\mathsf{app}(v_1, \mathsf{error}) \rightarrow \mathsf{error}}$$

Obsérvese que error no es un valor, esto es necesario para evitar ambigüedades en la evaluación, si error fuera declarado como valor entonces el programa $(\lambda x : \mathsf{Nat}.0)\mathsf{error}$ se reduciría por un lado a 0 y por otro a error.

La semántica estática se extiende mediante:

$$\frac{}{\Gamma \vdash \mathsf{error} : \mathsf{T}}$$

Esta regla permite que error reciba cualquier tipo, de manera que la propiedad de unicidad[1] de tipos se pierde. Esto es necesario para preservar los tipos.
Por ejemplo los términos $(\lambda x : \mathsf{Bool}.x)\mathsf{error}$ y $(\lambda x : \mathsf{Bool}.x)(\mathsf{error\ true})$ podrían surgir en un mismo programa de manera que error debe tener tipos Bool y Bool $\rightarrow$ Bool respectivamente.

La propiedad de unicidad de tipos no es importante desde el punto de vista teórico pero sí causa problemas prácticos para los algoritmos de inferencia de tipos. Una idea tentadora para recuperar esta propiedad consiste en anotar a la expresión error con un tipo adecuado. De esta manera la regla anterior se convertiría en

$$\frac{}{\Gamma \vdash \mathsf{error}_{\mathsf{T}} : \mathsf{T}}$$

Sin embargo obsérvese lo siguiente:

$$(\lambda x : \mathsf{Nat}.x)\big((\lambda y : \mathsf{Bool}.5)\mathsf{error}_{\mathsf{Bool}}\big) \rightarrow (\lambda x : \mathsf{Nat}.x)\mathsf{error}_{\mathsf{Bool}}$$

Y la preservación de tipos se ha destruido!!! De manera que debemos conformarnos con perder la unicidad de tipos y usar una misma expresión error con diversos tipos.

La propiedad de preservación se enuncia de la misma manera que antes pero, dado que la expresión error es una forma normal que no es un valor, el progreso del sistema debe reformularse como sigue:

[1] Cada expresión tiene a lo más un tipo, es decir que si una expresión se puede tipar entonces su derivación formal es única al igual que el tipo derivado.

Proposición 1 (Progreso) *Si* $\vdash e : \mathsf{T}$ *entonces sucede una y sólo una de las siguientes situaciones:*

1. e *es un valor.*
2. *Existe* e' *tal que* $e \to e'$.
3. $e = \mathsf{error}$.

A continuación mejoramos el uso de excepciones al no abortar completamente la ejecución de un programa y manejar las excepciones de manera que el control del programa de un salto.

3. Manejo de Excepciones

En nuestro modelo anterior cualquier expresión donde error figure causa que el programa se detenga totalmente devolviendo a error como resultado de la evaluación. Formalmente esto significa que el error se propaga totalmente en la pila de ejecución, lo cual corresponde en la realidad a eliminar todos los records de activación hasta vaciar la pila.
Otra posibilidad, menos radical y más útil, consiste en instalar *manejadores de excepciones* de manera que al surgir un error, se lanza una excepción que causa la eliminación de los records de activación de la pila hasta hallar un manejador al cual se le transfiere el control del programa, es decir, este manejador dicta cómo continua la evaluación. En este sentido una excepción funciona como un salto o transferencia no local del control hasta el manejador más cercano en la pila. Esto se conoce como la técnica de atravesar la pila[2]. Definimos ahora un lenguaje con una forma simple de manejo de excepciones.

La sintaxis concreta se extiende mediante:

$$e ::= \ldots \mid \mathsf{error} \mid \mathsf{try}\ e_1\ \mathsf{catch}\ e_2$$

La sintaxis abstracta es:

$$e ::= \ldots \mid \mathsf{error} \mid \mathtt{catch}(e_1, e_2)$$

La nueva expresión $\mathtt{catch}(e_1, e_2)$se evalua como sigue: se intenta evaluar e_1 y si esta evaluación termina normalmente se devuelve su valor el cual será también el valor de la expresión try . . . catch. Si la evaluación de e_1 es anormal surgirá un error y se procederá a evaluar e_2 cuyo valor final será también el valor de la expresión try . . . catch. Esto se formaliza mediante las siguientes reglas de la semántica operacional estructural:

$$\frac{e_1 \to e_1'}{\mathtt{catch}(e_1, e_2) \to \mathtt{catch}(e_1', e_2)}$$

$$\frac{}{\mathtt{catch}(v, e_2) \to v} \qquad \frac{}{\mathtt{catch}(\mathsf{error}, e_2) \to e_2}$$

La semántica estática está dada por la siguiente regla:

[2]En inglés *stack unwinding*

$$\frac{\Gamma \vdash e_1 : \mathsf{T} \qquad \Gamma \vdash e_2 : \mathsf{T}}{\Gamma \vdash \mathtt{catch}(e_1, e_2) : \mathsf{T}}$$

Se observa que e_1, e_2 y $\mathtt{catch}(e_1, e_2)$ deben tener todas el mismo tipo puesto que la evaluación del `catch` podría devolver indistintamente e_1 ó e_2.

Hemos dado una idea de como funciona el manejo de excepciones con respecto a la pila de control a continuación formalizamos esta idea mediante el uso de la máquina $\mathcal{K}$.

3.1. Extensión de la máquina $\mathcal{K}$

Extendemos la máquina $\mathcal{K}$ para modelar la idea de atravesar la pila hasta hallar un manejador. Este proceso sucede como sigue:

- La evaluación de un `catch` instala un manejador en la pila de control.
- La evaluación de un error atraviesa la pila eliminando marcos hasta hallar el manejador más cercano.
- La evaluación continúa según lo dictado por este manejador, el cual utiliza la misma pila de control de manera que si hay errores en él se propagarán de la misma manera.

Introducimos ahora una nueva clase de estado, denotado $\mathcal{P} \prec\!\!\prec \mathsf{error}$ cuya intuición es devolver y propagar un error en la pila $\mathcal{P}$, en búsqueda del manejador más cercano. Un estado de la forma $\Box \prec\!\!\prec \mathsf{error}$ se considera final y corresponde al caso en que el error se propagó hasta vaciar totalmente la pila y abortar totalmente la ejecución del programa.

La categoría de marcos se extiende con uno que indica el cómputo pendiente al evaluar la expresión e_1, es decir un manejador:

$$\frac{}{\mathtt{catch}(-, e_2) \;\; \mathsf{marco}}$$

A continuación enunciamos y explicamos las nuevas transiciones:

- Si se quiere evaluar un error entonces se causa el efecto de propagarlo en la pila.

$$\frac{}{\mathcal{P} \succ \mathsf{error} \longrightarrow_{\mathcal{K}} \mathcal{P} \prec\!\!\prec \mathsf{error}}$$

- La evaluación de un manejador $\mathtt{catch}(e_1, e_2)$ se inicia al evaluar e_1.

$$\frac{}{\mathcal{P} \succ \mathtt{catch}(e_1, e_2) \longrightarrow_{\mathcal{K}} \mathtt{catch}(-, e_2) \,;\, \mathcal{P} \succ e_1}$$

- Si la evaluación de e_1 es exitosa al devolver su valor v_1 se elimina el marco del manejador y el control del programa continua sin salto.

$$\frac{}{\mathtt{catch}(-, e_2) \,;\, \mathcal{P} \prec v \longrightarrow_{\mathcal{K}} \mathcal{P} \prec v}$$

- Propagar un error en una pila $m\,;\mathcal{P}$ consiste en eliminar el marco tope m y propagar el error en $\mathcal{P}$, siempre y cuando m no represente el cómputo pendiente de un manejador.

$$\frac{m \neq \mathtt{catch}(-, e_2)}{m\,;\mathcal{P} \prec\!\!\prec \mathsf{error} \longrightarrow_{\mathcal{K}} \mathcal{P} \prec\!\!\prec \mathsf{error}}$$

- Propagar un error en la pila con tope $\mathtt{catch}(-, e_2)$ consiste en eliminar el manejador y continuar la evaluación con e_2

$$\frac{}{\mathtt{catch}(-, e_2)\,;\mathcal{P} \prec\!\!\prec \mathsf{error} \longrightarrow_{\mathcal{K}} \mathcal{P} \succ e_2}$$

Si se piensa en una implementación, el mecanismo de atravesar la pila es poco óptimo. Otra solución considera dos pilas: una de control y otra de manejadores, de manera que al lanzarse una excepción el manejador más reciente estará en el tope de la pila de manejadores y por lo tanto está disponible de inmediato. Se deja al lector dar los detalles de esta semántica.

3.2. Resolución del alcance

Es de importancia mencionar que el alcance en una expresión try . . . `catch` se resuelve de manera dinámica en contraste con el alcance en una expresión `let`, por ejemplo considérese el siguiente programa:

```
try
 let f = fun(x:Nat) => error
     g = fun(h:Nat->Nat) => try (h 0) catch 1
 in
  try (g f) catch 2
 end
catch
 3
```

Dada la resolución dinámica este código se evalua a 1, mientras que en el caso siguiente

```
let fail = 3 in
 let f = fun(x:Nat) => fail
     g = fun(h:Nat->Nat) => let fail = 1 in (h 0) end
 in
  let fail = 2 in (g f) end
 end
end
```

la evaluación final devuelve 3 puesto que la resolución del alcance de expresiones *let* es estática.

3.3. Extensión con referencias y excepciones

La semántica dinámica que permite extender el lenguaje nucleo imperativo con excepciones requiere de una generalización de la máquina $\mathcal{K}$ con memorias de forma que los estados son de la forma $\langle \mu, \mathcal{P}, e \rangle$ siendo μ una memoria, $\mathcal{P}$ una pila de control y e una expresión. De forma que las transiciones son de la forma

$$\langle \mu, \mathcal{P}, e \rangle \longrightarrow \langle \mu', \mathcal{P}', e' \rangle$$

Como antes el proceso de manejo de una excepción consiste en atravesar la pila hasta el punto donde se halle el manejador más cercano ($\texttt{catch}(-, e_2)$) y continuar ejecutando la expresión e_2. Sin embargo, existen dos maneras de definir la interacción entre referencias y excepciones:

- Semántica estandar: la memoria μ al iniciar la evaluación de e_2 es la misma que estaba en uso en el momento en que la excepción fue lanzada.
- Semántica transaccional: la memoria μ al iniciar la evaluación de e_2 aquella utilizada en el momento en que el manejador ($\texttt{catch}(-, e_2)$) fue instalado en la pila de control. Es decir, una excepción enrrolla la memoria deshaciendo cualquier efecto anterior. Esta es una semántica similar a la implementada en sistemas de bases de datos.

3.4. Uso de excepciones para optimizar programas

Las excepciones con manejo no solo sirven para manejar errores sino que pueden usarse para optimizar un programa por ejemplo para causar un corto circuito en la evaluación de expresiones profundamente anidadas. Como ejemplo considérese el siguiente programa que implementa la función de igualdad entre listas

```
fun lsteq(l1:[T],l2:[T]):Bool =>
  lsteq ([],[]) = true
| lsteq ((x:xs),(y:ys)) = x==y and lsteq(xs,ys)
| lsteq (_,_) = false
```

Si consideramos dos listas de longitud 1000000 tales que difieren únicamente en su último elemento el programa anterior necesitará liberar 100000 cómputos pendientes para `and` antes de contestar falso. Esto se debe entre otras cosas a que la función no es recursiva de cola.

En su lugar puede implementarse una función utilizando el poder del paradigma imperativo como sigue:

```
fun leqaux(l1:[T],l2:[T]): Void =>
  leqaux ([],[]) = void
| leqaux ((x:xs),(y:ys)) = if x == y then leqaux(xs,ys)
                           else error
| leqaux (_,_) = error


fun lsteq(l1:[T],l2:[T]):Bool =>
 try (leqaux (l1,l2));true catch false
```

En este caso al lanzar una excepción error no hay necesidad de liberar cómputos pendientes o atravesar la pila pues se devolverá `false` de inmediato.

4. Excepciones con valor

En esta sección estudiamos la forma más general del manejo de errores. Además de lanzar una excepción al surgir un error y permitir un salto en el control del programa nos gustaría que dicha excepción llevara consigo cierta información, por ejemplo si se lanza una excepción de error de sintaxis en un analizador sintáctico la excepción podría reportar este hecho enviando un mensaje al usuario. A continuación tratamos esta forma general de excepciones.

La sintaxis concreta se extiende como sigue:

$$e ::= \ldots \mid \mathsf{raise}\ e \mid \mathsf{handle}\ e_1\ \mathsf{with}\ x \Rightarrow e_2$$

La sintaxis abstracta se extiende como sigue:

$$e ::= \ldots \mid \mathtt{raise}(e) \mid \mathtt{handle}(e_1, x.e_2)$$

La semántica dinámica es similar a la del lenguaje anterior sólo que error se sustituye por $\mathtt{raise}(e)$ cuyo argumento se evalua antes de propagarse para determinar el valor que recibirá el manejador respectivo. Además la instruccion $\mathtt{handle} \ldots \mathsf{with}$ presenta un ligado el cual servirá para recibir el valor de una excepción. La semántica operacional estructural es:

$$\frac{e \to e'}{\mathtt{raise}(e) \to \mathtt{raise}(e')} \qquad \frac{}{\mathtt{raise}(\mathtt{raise}(v)) \to \mathtt{raise}(v)}$$

$$\frac{e_1 \to e_1'}{\mathtt{handle}(e_1, x.e_2) \to \mathtt{handle}(e_1', x.e_2)}$$

$$\frac{}{\mathtt{handle}(v_1, x.e_2) \to v_1} \qquad \frac{}{\mathtt{handle}(\mathtt{raise}(v_1), x.e_2) \to e_2[x := v_1]}$$

Además se debe propagar la excepción en todos los demás constructores, por ejemplo para la aplicación tenemos:

$$\frac{}{\mathtt{app}(\mathtt{raise}(v_1), e_2) \to \mathtt{raise}(v_1)} \qquad \frac{}{\mathtt{app}(v_1, \mathtt{raise}(v_2)) \to \mathtt{raise}(v_2)}$$

4.1. Extensión de la máquina $\mathcal{K}$

La categoría de marcos se extiende con

$$\frac{}{\mathtt{raise}(-)\ \ \mathsf{marco}} \qquad \frac{}{\mathtt{handle}(-, x.e_2)\ \ \mathsf{marco}}$$

Las nuevas transiciones son:

- Evaluación de un lanzamiento

$$\frac{}{\mathcal{P} \succ \texttt{raise}(e) \longrightarrow_{\mathcal{K}} \texttt{raise}(-) ; \mathcal{P} \succ e}$$

- El valor de la excepción se devuelve al lanzador `raise`, el cual se propaga. Es decir, el valor que la excepción envia se encapsula en el lanzador `raise`.

$$\frac{}{\texttt{raise}(-) ; \mathcal{P} \prec v \longrightarrow_{\mathcal{K}} \mathcal{P} \prec\!\!\prec \texttt{raise}(v)}$$

- Evaluación de un manejador

$$\frac{}{\mathcal{P} \succ \texttt{handle}(e_1, x.e_2) \longrightarrow_{\mathcal{K}} \texttt{handle}(-, x.e_2) ; \mathcal{P} \succ e_1}$$

- Devolución del valor de un manejador

$$\frac{}{\texttt{handle}(-, x.e_2) ; \mathcal{P} \prec v \longrightarrow_{\mathcal{K}} \mathcal{P} \prec v}$$

- Propagar un valor de excepción en una pila $m ; \mathcal{P}$ consiste en eliminar m y propagarlo en $\mathcal{P}$, siempre y cuando m no sea un manejador.

$$\frac{m \neq \texttt{handle}(-, x.e_2)}{m ; \mathcal{P} \prec\!\!\prec \texttt{raise}(v) \longrightarrow_{\mathcal{K}} \mathcal{P} \prec\!\!\prec \texttt{raise}(v)}$$

- Propagar un valor de excepción $\texttt{raise}(v)$ en la pila con tope $\texttt{handle}(-, x.e_2)$ consiste en la captura de la excepción por el manejador e implica eliminar a éste del tope de la pila y pasar el valor guardado en la excepción para continuar con la evaluación de $e_2[x := v]$

$$\frac{}{\texttt{handle}(-, x.e_2) ; \mathcal{P} \prec\!\!\prec \texttt{raise}(v) \longrightarrow_{\mathcal{K}} \mathcal{P} \succ e_2[x := v]}$$

5. Acerca del paso de valores de excepción

Existen dos maneras de propagar un valor de excepción, la que hemos elegido aquí que consiste en encapsular el valor en el lanzador propagando el lanzador completo $\texttt{raise}(v)$, o bien propagando simplemente el valor de excepción v, para lo cual la máquina $\mathcal{K}$ se comportaría como sigue:

$$\frac{}{\texttt{raise}(-) ; \mathcal{P} \prec v \longrightarrow_{\mathcal{K}} \mathcal{P} \prec\!\!\prec v}$$

$$\frac{}{\texttt{handle}(-, x.e_2) ; \mathcal{P} \prec\!\!\prec v \longrightarrow_{\mathcal{K}} \mathcal{P} \succ e_2[x := v]}$$

$$\frac{m \neq \texttt{handle}(-, x.e_2)}{m ; \mathcal{P} \prec\!\!\prec v \longrightarrow_{\mathcal{K}} \mathcal{P} \prec\!\!\prec v}$$

Ambas maneras son claramente equivalentes, sin embargo elegimos usar la primera pues el hecho de propagar $\texttt{raise}(v)$ nos permite mantener una analogía entre `raise` y la forma primitiva de tratar excepciones con la constante error. Así como mantener encapsulados los valores de excepción.

5.1. Acerca de la semántica estática

Puesto que el lanzamiento de una excepción $\texttt{raise}(e)$ surge en cualquier contexto su tipo deber ser arbitrario. Adicionalmente el valor de excepción puede ser cualquiera así que el tipo de e también es arbitrario y sin relación directa con el anterior.

$$\frac{\Gamma \vdash e : \mathsf{S}}{\Gamma \vdash \texttt{raise}(e) : \mathsf{T}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T} \quad \Gamma, x : \mathsf{S} \vdash e_2 : \mathsf{T}}{\Gamma \vdash \texttt{handle}(e_1, x.e_2) : \mathsf{T}}$$

Sin embargo estas reglas no son satisfactorias desde el punto de vista del proceso de inferencia de tipos al no haber relación entre S y T la declaración $x : \mathsf{S}$ es díficil de encontrar. Lo usual es usar un tipo para excepciones así como una declaración específica de excepciones con valores de un tipo dado. Discutimos a continuación esta idea.

5.2. Declaración de excepciones

En los lenguajes de programación reales existen varias clases de error estándar, como las mencionadas al inicio de esta nota, de manera que para hacer más legible un programa es deseable tener una expresión particular para cada error, por ejemplo `DivCero`, `FueradeRango`, `ArchivoNoEncontrado`, `SinMemoria`, etc. Estas son excepciones predefinidas que indican condiciones específicas de error y si bien podrían usarse para otro propósito particular, esto no es aconsejable. En su lugar un lenguaje de programación real permite introducir excepciones mediante una declaración directa dada por el usuario. De esta manera podemos asociar excepciones específicas con programas particulares facilitando el proceso de rastreo del punto de error.

A continuación describimos sin detalles las principales ideas de esta extensión:

- Declaración de excepciones: se introduce una categoría de etiquetas de excepción mediante una declaración de la forma

  $$\mathsf{exception}\ \ell\ \mathsf{of}\ \mathsf{T}$$

  Este mecanismo introduce un tipado en la etiqueta de excepción de la forma $\ell : \mathsf{T} \to \mathsf{TExn}$ donde donde TExn es un nuevo tipo asociado a las excepciones. En la última sección de esta nota hablaremos más de este tipo.

- Expresiones: las mismas que antes más la aplicación de una etiqueta de excepción:

  $$e ::= \ldots \mid \ell\, e \mid \texttt{raise}(e) \mid \texttt{handle}(e_1, x.e_2)$$

- Semántica estática:

  $$\frac{\Gamma \vdash e : \mathsf{TExn}}{\Gamma \vdash \texttt{raise}(e) : \mathsf{T}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T} \quad \Gamma, x : \mathsf{TExn} \vdash e_2 : \mathsf{T}}{\Gamma \vdash \texttt{handle}(e_1, x.e_2) : \mathsf{T}}$$

  $$\frac{\ell : \mathsf{T} \to \mathsf{TExn} \quad \Gamma \vdash e : \mathsf{T}}{\Gamma \vdash \ell\, e : \mathsf{TExn}}$$

- Semántica dinámica: la misma que antes.

5.3. Acerca del ligado en un manejador

En nuestro lenguaje un manejador es de la forma handle e with $x \Rightarrow e$ siendo x una variable de tipo TExn. Pero obsérvese que las únicas expresiones del tipo TExn son aquellas generadas por las etiquetas de excepció. En lenguajes de programación reales es más común que el manejo se defina mediante una caza de patrones de acuerdo a dichas etiquetas. Por ejemplo

```
exception CodErr of Nat
exception MalaSuerte    -- tal caso se considera como exception MalaSuerte of Void

fun f(x:Nat):Nat =>
  | x == 0 = raise (CodErr 0)
  | x < 13 = raise (CodErr (x-7))
  | x == 13 = raise MalaSuerte
  | x > 13 = (x-2) mod 4

...

handle (f 10) with CodErr 0 => 0
                  | CodErr 1 => 1
                  | CodErr x => x+8
                  | MalaSuerte => write (``Horror!!'')
```

Nuestra forma de definir los manejadores es más simple de tratar formalmente y no pierde esta generalidad pues el código anterior puede considerarse como azucar sintáctica para:

```
handle (f 10) with x => exncase x of
                          CodErr 0 => 0
                        | CodErr 1 => 1
                        | CodErr x => x+8
                        | MalaSuerte => write (``Horror!!'')
                          ...
```

donde el operador `exncase` es un operador de análisis de casos sobre el tipo de las excepciones.

De manera general, una expresión de la forma:

```
handle e with   ℓ1 e1 => e1'
              | ℓ2 e2 => e2'
              | ⋮
              | ℓk ek => ek'
```

es azucar sintáctica para:

```
handle e with x => exncase x of
                     ℓ1 e1 => e1'
                   | ℓ2 e2 => e2'
                   | ⋮
                   | ℓk ek => ek'
                   | otherwise => raise x
```

6. Acerca del tipo TExn

De acuerdo a la regla de tipado de `handle`$(e_1, x.e_2)$, la variable ligada x debe ser del tipo TExn. Este tipo denota a los valores de excepción pero ¿Cómo es su estructura interna? al respecto existen varias alternativas:

- Podemos definir $\mathsf{TExn} =_{def} \mathsf{Nat}$, de manera que cada excepción será un número obtenido mediante una convención similar a la usada en los sistemas Unix (`errno`), es decir una excepción es simplemente un código de error.
- Podemos definir $\mathsf{TExn} =_{def} \mathsf{String}$ de manera que cada excepción es simplemente un mensaje que reporta que algo inesperado ocurrio. El precio es que un manejador de errores tendría que hacer un análisis sintáctico del mensaje particular para poder proceder con la evaluación.
- De manera más conveniente para un código legible podemos pensar en que TExn es un tipo variante donde cada opción es una excepción particular. Por ejemplo:

$$\mathsf{TExn} =_{def} [DivCero : (),\ FueradeRango : (),\ ArchNoEnc : \mathsf{String}, Insufic : \mathsf{Nat}, \ldots, \ell_n : \mathsf{T}_n]$$

En este caso las excepciones son informativas y pueden llevar consigo distintos tipos de información. El problema es que un tipo variante se declara estáticamente en una implementación particular lo cual implica que el conjunto de excepciones está previamente definido por lo que el programador no tiene la capacidad de declarar excepciones nuevas lo cual es inaceptable.

6.1. TExn como tipo extensible

Para permitir la declaración de excepciones se puede utilizar un tipo variante extensible, tal y como se implementa en el lenguaje ML. En este sentido la declaración de una excepción mediante la instrucción exception ℓ of T debe entenderse como sigue: si antes de la declaración se tiene $\mathsf{TExn} = [\ell_1 : \mathsf{T}_1, \ldots, \ell_n : \mathsf{T}_n]$ entonces la definición de excepción causa que se verifique si la etiqueta ℓ es distinta a cualquier etiqueta ya presente $\ell_1, \ldots, \ell_n$ en cuyo caso se extiende el tipo excepción quedando como $\mathsf{TExn} = [\ell_1 : \mathsf{T}_1, \ldots, \ell_n : \mathsf{T}_n,\ \ell : \mathsf{T}]$.