Exception Handling in MiniML

Final Project Report for CS1023
Software Development Fundamentals

Submitted by:

Devansh Tripathi
CS24BTECH11022

Date: May 2, 2025

Indian Institute of Technology Hyderabad

1 Introduction

1.1 MiniML

MiniML is an implementation of an eager statically typed functional language with a compiler and abstract machine.
It has the following constructs:

- Integers with arithmetic operations `+, -` and `*`.
- Since there is no exceptions defined in the language by default, there is no divison operation.
- Booleans with conditional control flows and integer comparisons (`=,<`).
- Recursive functions and function application.
- Toplevel definitions.

1.2 Aim of the Project

The main aim of this project is to extend the functionality of the MiniML language, and add exceptions and exception handling.

2 Features

The language now supports integer division.
Additionally, two new constructs have been added to the language:

- `raise`: Used to raise exceptions.
- `try-with`: Used to handle exceptions.

The exceptions that have been taken into consideration are:

- `DivisionByZero`: Raised by the machine when the divisor in a division operation is zero.
- `GenericException of int`: A Generic Exception type that can be used to define custom exceptions. Takes an integer code to represent the exception type.

Along with this, type checking has also been modified so that the incorrect datatypes are handled at runtime rather than before execution. To implement this, the definition of closures has also been modified.

3 Type System

The type checking system has been changed to accommodate incorrect datatype handling in runtime rather than before execution. The changes and additions made are as follows:

- Added a new `exptn` type in syntax for exceptions.
  ```
  type exptn =
  | DivisionByZero
  | GenericException of int
  ```
- Consequentially, syntax type `TExptn` and machine value type `MExptn` have also been added.
- Modified `type_check.ml` to enable bypass the incorrect data type error during runtime.
  Example: `| Plus (a, b) -> (try check ctx TInt a; check ctx TInt b; TInt with Type_Error -> TExptn)`
  Here, the types of `a` and `b` are checked, and if any of them are not `TInt`, then instead of stopping there, we pass `TExptn` as the return type instead, and the incorrect datatypes are handled in `machine.ml`.
- Incorrect datatype handling example in `machine.ml`:
  ```
  let add = function
  | (MInt x) ::  (MInt y) ::  s -> MInt (y + x) ::  s
  | _ -> [MExptn (Syntax.GenericException (-1))]
  ```
  Here, the `add` function will only return the sum of `x` and `y` if both of them are of the type `MInt`. Otherwise it throws the error `GenericException -1`.
- To do this for user-defined functions, the definition of `MClosure` has been changed.
  `MClosure of name * frame * environ * Syntax.ty`
  `MClosure` now also stores the datatype required for the parameter, and this is checked at runtime. Here, the exception thrown is `GenericException 1`.
- For the `try-with` block, the type checking is done as follows:
  - First, the type of the test expression is checked and returned if it is not of the type `TExptn`.
  - If it is of type `TExptn`, then the type of the last expression in the list of expressions in the `with` block is given.
- Note the for division by zero cases, the return type is still `int`, as the return type is always `TInt` (I was unable to access the value of the divisor in type checking so I was forced to keep the return type as `int`).

4 Exception Handling

4.1 Exceptions

The exceptions that can be thrown by the machine are:

- `DivisionByZero`: Thrown when the divisor of a division is zero, eg. `5/0`.
- `GenericException -1`: Thrown when the operands of an arithmetic or logical operation are incorrect, eg. `5+true`, `false<4`.
- `GenericException 1`: Thrown when the function parameter has incorrect data type, eg. `fact true` where `fact n` returns the factorial of a positive integer `n`.

These exceptions are detected in the following ways:

- `DivisionByZero`: The topmost value in the value stack, i.e., the divisor is checked if it is zero.
```
let div = function
| (MInt 0) ::  (MInt _) ::  s -> MExptn Syntax.DivisionByZero::
s
| (MInt x) ::  (MInt y) ::  s -> MInt (y/x) ::  s
| _ -> [MExptn (Syntax.GenericException (-1))]
```
- `GenericException -1`: The datatypes of the machine values of the operands are checked and this exception is thrown when they are not the expected types (demonstrated above).
- `GenericException 1`: The datatype of the function parameter is checked before executing `ICall` further:
  - The definition of `MClosure` has been modified to also include the type of the parameter expected.
  - We use a helper function `get_type` which returns the corresponding machine value type of its argument.
  - If it does not match. then the exception is thrown. else the execution continues.

Exceptions can also be raised manually by the user by using the `raise` command.
`raise DivisionByZero;;` will throw a `DivisionByZero` exception.

4.2 Exception Handling

To handle the exceptions thrown by the machine, the `try-with` construct has been added. The general syntax of this construct is:

```
try{test-expression} with {| error1 -> exp1 | error2 -> exp2 ....}
```

The multiple error cases have been implemented by splitting the expression inside the `with` block as a list of `cases` in `parser.mly`:

```
| TRY e = expr WITH LBRACE cases = nonempty_list(case) RBRACE
{ Try (e, cases) }
```

where `case` is parsed as:

```
case:
| PIPE e=exptn TARROW e1=expr
(e, e1)
```

`PIPE`, `LBRACE`, `RBRACE`, `TRY` and `WITH` are user-defined tokens for `|`, {, }, `try and with` respectively.

Next, the test expression is executed, after which it is type-checked. If its type is `TExptn`, then we recursively go through the types of the expressions in the `with` block:

```
| Try (e, cases) ->let ty = type_of ctx e in
(* Recursively go through the list and return the type of the last
expression.  *)
let rec match_cases cases exp_case = match cases with
| (_, exp) ::  body -> let t' = type_of ctx exp in match_cases body
t'
| [] -> exp_case in
(* Assign the return type to t' if the test expression is an exception.
*)
let t' = match_cases cases ty in if ty = TExptn then t' else ty
```

It can be noted though, that this approach has two main drawbacks:

- Since division by zero does not have a `TExptn` type after type checking, it will have a type `TInt` and so any `with` expressions having `DivisionByZero` must have their expression as type `int` if the test expression has that error, otherwise an error will be raised.
- If different cases have different types, then the return type would be that of the last case, which can lead to confusion in the output.

Finally, in `machine.ml`, the topmost value in the stack is obtained, and we recursively iterate through our case list to match it with the different error cases. If we find a match, we execute the expression under it, or else we leave the values as they are, which means that either there was no error produced or the produced error has not been covered in the `with` block.

5 Example Cases

5.1 Raising Exceptions

Let us see how and when exceptions are raised:

- Division by zero:
```
miniML> 1/0;;
- :  int = Division By Zero
```
- Wrong operands:
```
miniML> 1+true;;
- :  error = Generic Exception -1
```
- Wrong function parameter:
```
miniML> let double = fun f (n:int) :  int is 2*n;;
double :  int -> int = <fun>
miniML> double 5;;
- :  int = 10
miniML> double true;;
- :  error = Generic Exception 1
```
- Using `raise`:
```
miniML> raise DivisionByZero;;
- :  error = Division By Zero
miniML> raise GenericException 42;;
- :  error = Generic Exception 42
```

5.2 `try-with` Usage

Next, let us see use cases of the `try-with` block:

- Simple Exception Handling (`double` is the same function as defined above):
```
miniML> try {21/6} with {|DivisionByZero -> 0};;
- :  int = 3
miniML> try {21/0} with {|DivisionByZero -> 0};;
- :  int = 0
miniML> try {2 + false} with {|GenericException -1 -> true};;
- :  bool = true
miniML> try {2 + false} with {|GenericException -1 -> 4};;
- :  int = 4
miniML> try {double true} with {|GenericException 1 -> 10};;
- :  int = 10
```
- Multiple Cases (`double` is the same function as defined above):
```
miniML> try {21/0} with {|DivisionByZero -> 10 | GenericException
-1 -> 20 | GenericException 1 -> 30};;
- :  int = 10
```

```
miniML> try {10 + false} with {|DivisionByZero -> 10 | GenericException
-1 -> 20 | GenericException 1 -> 30};;
- :  int = 20
miniML> try {double true} with {|DivisionByZero -> 10 | GenericException
-1 -> 20 | GenericException 1 -> 30};;
- :  int = 30
```

- Unhandled Exceptions:
  ```
  miniML> try {3/0} with {|GenericException 1 -> 9};;
  - :  int = Division By Zero
  miniML> try {4/false} with {|DivisionByZero -> 0 |GenericException
  1 -> 10};;
  - :  int = Generic Exception -1
  ```

- Nested Blocks:
  ```
  miniML> try {4 + try{4/0} with {|DivisionByZero -> false}} with
  {|GenericException -1 -> 140};;
  - :  int = 140
  ```

5.3 Limitations

However, there are multiple cases where this implementation fails or produces unexpected results:

- Non-exception value raised:
  ```
  miniML> raise 5;;
  Syntax error at line 0, characters 6-7:
  syntax error
  ```
  This is because `raise` is not type checked due to complications of its argument being an explicit `exptn` and not included in the `expr` group which is type-checked.

- Multiple Same Exception Cases:
  ```
  miniML> try {2/0} with {|DivisionByZero -> 0 |DivisionByZero ->
  5};;
  - :  int = 0
  ```
  The first expression is always executed.

- Different Data Types of Cases:
  ```
  miniML> try {3+true} with {|GenericException -1 -> 5 |DivisionByZero
  -> false};;
  - :  bool = 5
  ```
  Since the return data type in the type checker is determined by the last expression in the list, the datatype of the result can be printed incorrectly if the datatypes of the cases are different.

- Fixed Return type of Division by Zero cases:
  ```
  miniML> try {5/0} with {|DivisionByZero -> false};;
  ```

```
- :  int = false
```
The divisor's value cannot be accessed in the type checking process so we need to fix its return type as `int` and treat it as general division, which may print the datatype of the result incorrectly.

- Handling Raised Exceptions:
```
miniML> try {raise DivisionByZero} with {|DivisionByZero -> 0};;
- :  int = Division By Zero
miniML> try {raise GenericException 1} with {|GenericException
1 -> true};;
- :  bool = Generic Exception 1
```
The exceptions are compared by their string values, which is quite unreliable in these cases. However, when I tried to implement direct matching, it lead to other errors, so I had to keep this one.

6 Challenges

The main challenges encountered in this project were:

- Understanding how the parsing of expressions works.
- Adding a totally new data type `exptn`.
- Adding new constructs.
- Bypassing the type-checking restrictions to implement custom exceptions.

7 How to Use

- Make sure **OCaml** has been installed and the **OCaml Development Environment** has been set up.
- Activate the **opam switch**
- Clean the previous build with
`dune clean`
- Build the executable again with
`dune build src/miniml`
- You can now run `miniml.exe` to use the top level and see how exceptions and ther handling behave.

8 Conclusion

This project successfully extends MiniML with exception handling capabilities through the addition of new constructs and runtime type checking. While the current implementation has certain limitations, it demonstrates how core programming language features can be enhanced within a functional language compiler. Future work could include improving pattern matching on exceptions and enhancing type inference in try-with blocks.