universität innsbruck

WS 2025/2026


# Advanced Functional Programming

## Week 9 – System Programming, Exceptions

René Thiemann

Department of Computer Science

# Last Week

- applicative functors and applicative style parsers
- monad transformers
- exercise on lexicographic path order (LPO)
  - LPO is parametrized by precedence $p : \Sigma \to \mathbb{N}$
    - $\dfrac{s_i \succeq_{LPO} t}{f(s_1, \dots, s_n) \succ_{LPO} t}$ (sub)
    - $\dfrac{s_i \succ_{LPO} t_i \qquad s \succ_{LPO} t_{i+1} \qquad \dots \qquad s \succ_{LPO} t_n}{s = f(s_1, \dots, s_{i-1}, s_i, s_{i+1}, \dots, s_n) \succ_{LPO} f(s_1, \dots, s_{i-1}, t_i, t_{i+1}, \dots, t_n)}$ (lex)
    - $\dfrac{p(f) > p(g) \qquad s \succ_{LPO} t_1 \qquad \dots \qquad s \succ_{LPO} t_n}{s = f(\dots) \succ_{LPO} g(t_1, \dots, t_n)}$ (prec)
  - task: find precedence such that $\ell \succ_{LPO} r$ for all rules of a TRS or fail
  - task is NP-complete, positive answer ensures termination of TRS
  - input: `String`
  - using ARI parser: `[Rule]`
  - using LPO encoder: `String` (SMT encoding)

# System Programming

# Current Situation

- given TRS, we obtain some SMT-Lib encoding such as

```
(set-logic QF_LIA)
(declare-fun x1 () Int)
(assert (and (<= 1 x1) (<= x1 4)))
...
(assert (= x7 (or (and (> x5 x2) x6) x4)))
(assert x7)
(assert (> x1 x5))
(check-sat)
```

- an SMT solver takes this as input, and either reports unsatisfiability or provides a model, i.e., concrete numbers and Boolean values for each `xi`
- obvious question: how to invoke SMT solver from Haskell program?
- solution: use `System.Process`
- upcoming: a glimpse of system programming with Haskell, focussed on this application

# Communication via Files

- meta algorithm
  1. write SMT encoding into `file.smt2` (`writeFile`)
  2. invoke SMT solver on `file.smt2` to produce `answer.txt` (`createProcess`)
  3. read answer from file `answer.txt` (`readFile`)
  4. obtain sat/unsat from answer (==)
  5. in case a model was found, extract precedence from answer (Parsec)
  6. delete `file.smt2` and `answer.txt` (`removeFile`)
- details
  - as SMT solver we propose Z3 (`https://github.com/Z3Prover/z3`)
  - concrete problem 1: understand lazy I/O
  - concrete problem 2: how to choose filenames, where should files be stored
  - concrete problem 3: how to invoke external processes

# First Version of File Communication

```
commFile1 trs = do
  let smtFile = "file.smt2"
  let enc = snd $ lpoTrsEncoder trs
  writeFile smtFile enc
  let answerFile = "answer.txt"
  -- later: invoke "z3 -smt2 file.smt2 > answer.txt", now simulate by
  writeFile answerFile $ "sat\n" ++ concat (replicate 100000 "ab\n")
  answer <- readFile answerFile
  removeFile answerFile
  removeFile smtFile
  let (firstLine : rest) = lines answer
  result <- if firstLine == "sat"
    then return $ Just $ "parse " ++ show (length rest) ++ " lines"
    else return Nothing
  return result
```

# Concrete Problem 1: Understand Lazy I/O

```
commFile1 trs = do
  answer <- readFile answerFile
  removeFile answerFile
  ... answer ...
```

- in Haskell I/O is lazy
  - `answer <- readFile answerFile` immediately returns after its invocation without reading the full file
  - advantage of lazy I/O:
    ```
    do s <- readFile "input.txt"
       writeFile "output.txt" (map toUpper s)
    ```
    convert (large) file to upper-string with constant memory consumption
  - disadvantage: code might crash because of lazy I/O; consider variant
    ```
    do originalContent <- readFile "foo.txt"
       writeFile "foo.txt" "overwrite the content"
       return $ take 20 originalContent
    -- *** Exception: foo.txt: withFile: resource busy (file is locked)
    ```
- solution: fine-grained control with `Handle`s, force evaluation

## Handles

- in Haskell one can perform I/O via handles
- several I/O operations are actually done via handles

```
putStrLn :: String -> IO ()              -- print to stdout
hPutStrLn :: Handle -> String -> ()      -- print to handle
getLine :: IO String                     -- read from stdin
hGetLine :: Handle -> IO String          -- read from handle
...
stdin, stdout, stderr :: Handle
getLine = hGetLine stdin
...
```

- `stdin`, `stdout`, `stderr` are handles for text input and output, but one can also get handles in other ways (open file, open network connection, ...)
- common operations

```
h <- openFile fileName mode -- open file in ReadMode, WriteMode, ...
hClose h                    -- close handle
hFlush h                    -- flush buffered output
```

# Things to Know About Handles

- reading from a handle is done lazily
  - `s <- hGetContents h` and other read commands produce lazy strings: only when `s` is accessed, it is actually read from the handle
  - as soon as `hClose h` is invoked on some handle of an input stream, further read accesses result in exceptions
- example: the returned value is accessed after closing the handle

```
do h <- openFile "foo.txt" ReadMode
   s <- hGetContents h
   hClose h
   return $ take 20 s
-- *** Exception: foo.txt: ... delayed read on closed handle
```

- solution: enforce full evaluation of return value, e.g., via `($!!)` from DeepSeq package

```
   s <- hGetContents h
   result <- return $!! take 20 s   -- first 20 chars will be read
   hClose h
   return result
```

# Convenience Method for Doing File-I/O

- for the pattern "open a file – read/write something – close a file" there is special support by some higher order function

```
withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r
```

  - `withFile f m a` will open the file to get a handle `h`, execute action `a h`, and then close `h`
  - closing `h` file will be ensured, even if `a h` throws an exception
- example from previous slide in convenient form

```
withFile "foo.txt" ReadMode (\h -> do
   s <- hGetContents h
   return $!! take 20 s)
```

# Concrete Problem 2: Filenames

```
commFile1 trs = do let smtFile = "file.smt2"
  writeFile smtFile enc >> ... >> removeFile smtFile
```

- issue 1: `file.smt2` might already exist in filesystem and accidently gets overwritten
- issue 2: program is not thread-safe
  - running two instances of `commFile1` in parallel will result in problems
- solution: ask operating system for temporary file, given template name of type `String`

```
openTempFile :: FilePath -> String -> IO (FilePath, Handle)
withTempFile :: FilePath -> String -> (FilePath -> Handle -> IO a) -> IO a
emptyTempFile :: FilePath -> String -> IO FilePath     -- not opened
... -- variants which write in default temp-directory of OS
```

  - `FilePath` is directory where temporary file should be created
  - template name is expanded, e.g. "file.smt2" might turn to "file4Xa54.smt2"
  - generated filename and handle are made accessible
  - temporary files are opened in `ReadWriteMode`
  - the `withTemp...` variants additionally take care of deleting the temp-file after invocation

# Second Version of File Communication

```
commFile2 trs =
  withSystemTempFile "file.smt2" (\ smtFile hf ->
  withSystemTempFile "answer.txt" (\ answerFile ha -> do
    let enc = snd $ lpoTrsEncoder trs
    hPutStrLn hf enc
    hFlush hf

    -- TODO: invoke "z3 -smt2 smtFile > answerFile", or simulate by
    hPutStrLn ha $ "sat\n" ++ concat (replicate 100000 "ab\n")
    hSeek ha AbsoluteSeek 0

    answer <- hGetContents ha

    let (firstLine : rest) = lines answer
    result <- if firstLine == "sat"
      then return $!! Just $ "parse " ++ show (length rest) ++ " lines"
      else return Nothing
    return result  ))
```

# Concrete Problem 3: Creation of External Processes

- Haskell offers the following main function to invoke external processes

```
createProcess :: CreateProcess ->
   IO (Maybe Handle, Maybe Handle, Maybe Handle, ProcessHandle)
```

- `CreateProcess` is a record datatype with 15 fields to configure what process should be called in which way
  - one usually uses one of the following functions and overwrites specified entries

    ```
    proc :: FilePath -> [String] -> CreateProcess
    shell :: String -> CreateProcess
    ```

- `ProcessHandle` is a handle to control the new process

```
waitForProcess :: ProcessHandle -> IO ExitCode
terminateProcess :: ProcessHandle -> IO ()
```

- `Maybe Handle` provide access to stdin, stdout, stderr of the new process, which might also be setup via `CreateProcess`

# Final Version of File Communication

```
commFile3 trs = withSystemTempFile "file.smt2" (\ smtFile hf -> do
    answerFile <- emptySystemTempFile "answer.txt"  -- do not immediately open

    let enc = snd $ lpoTrsEncoder trs
    hPutStrLn hf $ enc ++ "(exit)\n"    -- tell z3 to terminate after search
    hClose hf                           -- flush and release write-lock on smtFile

    let cpConfig = shell $ "z3 -smt2 " ++ smtFile ++ " > " ++ answerFile
    (_,_,_,ph) <- createProcess cpConfig    -- start z3
    _ <- waitForProcess ph                  -- and wait until it has finished

    answer <- readFile answerFile
    let result = head (lines answer) == "sat"     -- no precedence extraction

    removeFile answerFile                                      -- cleanup

    return result  )                    -- result: does LPO exist for this TRS
```

# Limits of Current Workflow

- situation: two parties (Haskell HS, z3 solver), both accessing shared resources
- simple communication via files
  - HS writes smtFile and spawns solver
  - solver reads smtFile
  - solver writes answerFile and terminates
  - HS reads answerFile
  - HS prints result and terminates
- limitation: cannot model more complex scenarios, e.g., where HS issues commands to solver that depend on previous answers of solver
  - HS: solve these constraints
  - solver: "sat"
  - HS (after reading "sat"): give me the value of $x_1$ and $x_5$
  - solver: $x_1 = 5$, $x_5 = True$
  - HS: solve other constraints
  - solver: "unsat"
  - HS does not ask for values after reading "unsat" (query might even crash the solver)
  - ...

# Towards a More Complex Workflow

- in order to communicate with external processes, instead of files one can use pipes
- during process creation, one can setup communication channels via pipes

```
do let cpConfig = (proc "z3" ["-in"]){
        std_out = CreatePipe,
        std_in  = CreatePipe }
   (Just hSmtIn, Just hSmtOut, _, pHandle) <- createProcess cpConfig
```

  - command line argument `-in` tells z3 to take input from stdin
  - overwriting `cpConfig {std_in = CreatePipe}` tells `createProcess`, that Haskell program wants to have a handle to stdin of the spawned process, implemented by a pipe
  - `hPutStrLn hSmtIn "hello"` will send `"hello\n"` to new process
  - rule of thumb: after issuing a command to the solver, one should invoke `hFlush hSmtIn` to ensure that all buffers will be written
  - similarly, everything that the spawned process writes to stdout can be read via `hSmtOut`
  - question: how much should be read from the solver? depends on protocol!

# Communication with an Interactive Program such as z3

- after issuing the `(check-sat)` command, z3 will answer with `"sat\n"` or `"unsat\n"`
- if the answer was `"sat\n"`, one can issue a z3-command such as `(get-value (x1 x5))`
- afterwards, z3 will answer with `"((x1 2)(x5 7))"`
  (string might contain additional whitespace, including several newlines)
- task 1: write a parser for these kinds of answers, e.g., using Parsec
- task 2: invoke the parser
  - problem 1: how long should we read from `hSmtOut`?
    - obvious: until final closing ")" has been read
    - but to detect this final closing ")", we need to run the parser
  - problem 2: `runParser` (or `parse`) expects a `String` as input, not a `Handle`
  - solution: use lazy I/O
    - just pretend that one can read and get access to the full string that z3 will write to stdout during its invocation, by invoking `hGetContents hSmtOut`
    - stop consuming input after final closing ")"

# Parsing with Lazy I/O

- Haskell is surprisingly simple, but tricky

```
smtAnswerParser :: Parser [(String, Integer)]
smtAnswerParser = ... Exercise ...

-- h might be hSmtOut
smtAnswerFromHandle :: Handle -> IO [(String, Integer)]
smtAnswerFromHandle h = do
  input <- hGetContents h
  case parse smtAnswerParser "" input of
     Left e -> error $ show e
     Right res -> return res
```

- remarks
  - one needs to ensure that the parser immediately stops after reading the final closing ")"
  - for simplicity we assumed that we are only interested in integer values, but not in Booleans

# Full LPO-Solver

```
lpoSolver :: TRS -> IO (Maybe LPO)
lpoSolver trs = do
  let (precMap, smtString) = first M.toList $ lpoTrsEncoder trs
  let cpConfig = (proc "z3" ["-in"]){ std_out = CreatePipe, std_in = CreatePipe }
  (Just hSmtIn, Just hSmtOut, _, pHandle) <- createProcess cpConfig   -- start z3
  hPutStrLn hSmtIn smtString >> hFlush hSmtIn                      -- command: detect sat
  satStatus <- hGetLine hSmtOut                                    -- read sat/unsat line
  answer <- if satStatus /= "sat" then return Nothing else
    if null precMap then return $ Just $ LPO_with_Precedence []   -- special case
    else do hPutStrLn hSmtIn $ smtRequestValues (map snd precMap)
            hFlush hSmtIn
            parsedModel <- M.fromList <$> smtAnswerFromHandle hSmtOut
            return $ Just $ LPO_with_Precedence $
              map (\ (f, xi) -> (f, parsedModel M.! show xi)) precMap
  hPutStrLn hSmtIn "(exit)"                                        -- final cleanup: soft
  hClose hSmtOut >> hClose hSmtIn
  terminateProcess pHandle                                         -- or hard termination
  return $ answer                                             -- eventually return result
```

## Remarks

- special treatment for empty list is required, since z3 does not like to be asked for an empty list of values
- we first give z3 the chance to terminate itself via command `"(exit)"`, afterwards we use the harder `terminateProcess` method (SIGTERM signal, i.e., `kill`) (there are also variants to send a SIGKILL signal, i.e., `kill -9`)
- the design is not optimal, as the communication and the special treatment of empty list is implemented inside `lpoSolver`
  - problem: implementation needs to be repeated for every other z3-based search algorithm
  - solution: exercise

# Exceptions

## How to Handle Errors

- distinguish two kind of errors
- errors under control of programmer
  - how to handle parsing error?
  - how to handle division-by-zero when evaluating user provided expression?
  - how to handle invocation of function if input is invalid?
- errors not under our control
  - all kind of I/O errors: network, file not found, no write permission, external process crashes, ...
  - runtime errors that arise when invoking custom functions
- handling the former can be done using `Maybe`, `MonadError`, etc.; has been discussed thoroughly
- both kinds of errors can be handled via exceptions

# Exceptions

- exception handling is supported by several programming languages, including Haskell
- exceptions can be thrown by any function via one of these functions

```
error :: String -> a
throw :: Exception e => e -> a
throwIO :: Exception e => e -> IO a
```

- whether some function evaluation may result in an exception is not visible from its type
- `error` and `throw` are imprecise exceptions
  - pure value `(throw ex + error "fail") :: Int` may result in either of the exceptions
  - use `throwIO` for precise exceptions, e.g. `throwIO ex >> error "fail"` will result in `ex`
- exception handling can be done for errors that occur several layers down the call stack
- in Haskell, exceptions can only be catched within I/O-monad
  - reason: unspecified evaluation order, e.g., consider problem
    `let x = error "fail" in f (g x) (h x)`
    where both `g` and `h` are allowed to perform exception handling
- no special syntax for exception handling; instead: use functions

## Try

- in this part we are looking at `try` of `Control.Exception`, and not the `try` of `Parsec`!
- `try :: Exception e => IO a -> IO (Either e a)`
  - `try action` returns `Right x` if `action` results in `x` without raising an exception
  - `try action` returns `Left e` if `action` results in an exception of type `e`
- one often has to choose a concrete type `e` for `e` by a type annotation
- choosing `e = SomeException` catches all exceptions, since `SomeException` is the root of all exception types; usually, you should not catch all exceptions!
- consider the following code

```
badNumber, goodNumber :: Int
badNumber = 5 `div` 0
goodNumber = 5 `div` 1

tryBad, tryGood :: IO (Either SomeException ())  -- catch any exception
tryBad = try (putStrLn $ show badNumber)          -- Left divide by zero
tryGood = try (putStrLn $ show goodNumber)        -- 5, Right ()
```

- neither `tryBad` nor `tryGood` result in an exception

# Try and Laziness

- consider the following code (`e = SomeException` omitted)

```
tryReturnBad = try (return badNumber) >>= (\ x -> putStrLn $ show x)
```

- execution results in: `Right *** Exception: divide by zero`
- reason is lazy evaluation
  - `return badNumber` does not throw an exception, since evaluation of `badNumber` is not enforced at this point
  - hence, `try (return badNumber)` is equivalent to `return $ Right badNumber`
  - `x` is then bound to `Right badNumber`
  - `putStrLn $ show x` starts to print, where
    - first the string `"Right "` is produced
    - then `badNumber` is evaluated and an exception occurs
- solution: use `evaluate :: a -> IO a` instead of `return` to force evaluation to WHNF

```
tryEvaluateBad = try (evaluate badNumber) >>= (putStrLn . show)
```

  results in `Left divide by zero` where exception has been catched
- if WHNF is not enough for use-case, then replace `evaluate` by methods from DeepSeq module, e.g., `($!!)`

## Catching Exceptions

- use-case: deal with exception instead of returning `Either`-type
- most basic version: `catch :: Exception e => IO a -> (e -> IO a) -> IO a`
- behavior of `catch a h`
  - execute action `a`
  - if execution throws an exception `e`, then `h e` is executed
- example application

```
tryToRead f = catch (readFile f) $ \e ->
  do let err = show (e :: IOException)
     hPutStr stderr ("Warning: Couldn't open " ++ f ++ ": " ++ err)
     return ""
```

  - `IOException` is root of all I/O exceptions
  - hence, `tryToRead` catches I/O exceptions, but does not catch other exceptions, e.g., test `tryToRead $ "file" ++ show (1 `div` 0)`

## Catching Exceptions with Multiple Handlers

- use-case: deal with exception, choose handler depending on exception type
- obvious idea: nested `catch`-applications

```
f = expr `catch` \ (ex :: ArithException) -> handleArith ex
         `catch` \ (ex :: IOException)    -> handleIO    ex
```

- problem besides inefficiency
  - if first exception handler `handleArith` raises an `IOException`, then this is caught by the second handler
  - aim: select one exception handler depending on raised exception
- solution via `catches :: IO a -> [Handler a] -> IO a`

```
f = expr `catches`
   [Handler (\ (ex :: ArithException) -> handleArith ex),
    Handler (\ (ex :: IOException)    -> handleIO    ex)]
```

- interesting datatype for handlers
  - `data Handler a = forall e . Exception e => Handler (e -> IO a)`
  - `Handler a` does not depend on `e` because of usage of `forall`
  - hence, one can add exception handlers for different choices of `e` in the same list

## Catching Exceptions with Predicates

- use-case: select which exceptions to handle based on a predicate
- `catchJust :: Exception e =>`
  `(e -> Maybe b) -> IO a -> (b -> IO a) -> IO a`
  - the function `e -> Maybe b` selects if an exception `e` should be treated
  - if so (`Just b`), the handler is invoked, otherwise the exception will be left untouched
- examination of an `IOException`: consider module `System.IO.Error`
  - `type IOError = IOException`
  - `isPermissionError :: IOError -> Bool`
  - `isDoesNotExistError :: IOError -> Bool`
  - `isEOFError :: IOError -> Bool`
  - ...

# User-Defined Exception Types

- creating an exception type is easy; example

```
data MyException = NegativeInput | TooLarge deriving (Show)

instance Exception MyException   -- no methods required

easyPrimeTest, prime :: Integer -> Bool
easyPrimeTest x
 | x < 0 = throw NegativeInput
 | x > 30 = throw TooLarge
 | otherwise = x `elem` [2,3,5,7,11,13,17,21,23,29]

prime x = catchJust
  ( \ myE -> case myE of { TooLarge -> Just (); _ -> Nothing } )
  (evaluate $ easyPrimeTest x)
  (\ () -> error $ "TODO: run full prime test on " ++ show x)
```

## Literature

- Real World Haskell, Chapters 7, 19 and 20
  - Chapter 19 is partly outdated: describes no longer available `Exception` type, which was changed into an `Exception` class
  - Chapter 20 is partly outdated: uses deprecated `System.Cmd` and not `System.Process`
- https://hackage.haskell.org/package/base/docs/System-IO.html
- https://hackage.haskell.org/package/deepseq/docs/Control-DeepSeq.html
- https://hackage.haskell.org/package/temporary/docs/System-IO-Temp.html
- https://hackage.haskell.org/package/process/docs/System-Process.html
- https://hackage.haskell.org/package/base/docs/Control-Exception.html