Cloud Haskell Extension for Misty

Jakob Vokac

September 23, 2019

1 Introduction

This documentation covers the extended MISTY code for Cloud Haskell code generation, which is available on the folloring repository: `https://github.com/JakobVokac/MistyCH`. The code is an extension of the MISTY program, implemented by A. Arslanagic, which is available on:
`https://gitlab.com/aalen9/misty`.

Further documentation for MISTY in particular is located in the `MistyCH` folder, courtesy of A. Arslanagic.

2 Usage Instructions

The program, written in Haskell, requires Haskell Stack to run. Once Stack is installed on the computer, it is compiled by executing the following commands:

- `stack setup`
- `stack build`

This sets up the Haskell environment for the code.

Due to dependency constraints with Cloud Haskell, Misty and the Cloud Haskell output files have two different environments, both of which need to be built separately using the above instructions.

MISTY is located in the `MistyCH` folder and can be ran with `stack run` once in the folder with the environment built. One can also use the GHC interface by running `stack ghci`.

Once in the program, the user is asked to give input for which example to compile. These are the available options, which also include MISTY translation into Latex (which was the original purpose of MISTY):

- 0 - SimpleExample into Latex
- 1 - NamePassing into Latex
- 2 - SelectBranching into Latex
- 3 - RecursionEncoding into Latex
- 4 - SendingAbstractions into Latex
- 5 - BoolAbstraction into Latex
- 6 - BinaryOperation into Latex
- 7 - AccumulatingChannels into Latex
- 8 - MultipleInputs into Latex
- 9 - ApplyingAbstraction into Latex
- 0.1 - SimpleExample into Cloud Haskell
- 1.1 - NamePassing into Cloud Haskell
- 4.1 - SendingAbstractions into Cloud Haskell
- 5.1 - BoolAbstraction into Cloud Haskell
- 7.1 - AccumulatingChannels into Cloud Haskell
- 8.1 - MultipleInputs into Cloud Haskell

The Latex translation is outputted to the `TEXoutput` folder, which is located in the root of the project. The Haskell code is outputted to the `HSoutput` folder, also located in the root.

The Cloud Haskell output environment is located in the `HSoutput` folder. The code itself is located in the `mistyOutput` folder. Once the environment is built, the user can run the `stack ghci` command in the `mistyOutput` folder and load each example individually. The examples are executed by running `main`.

3 Code documentation

As mentioned before, the code is an addition to the MISTY program. The entirety of this extension is located in the `CloudHaskell` module. The modlue uses type definitions from the `Misty.Process`, `Misty.Types` and `Misty.Channel` modules. It uses the `Data.Char` module for checking for digits and it uses the `Data.List` module for list manipulation functions. It also makes use of the `Misty.Latex` module for the `ifProp` function as well as several `show` instances, which are convenient for printing certain data types. The `Debug.Trace` module is imported for debugging purposes.

3.1 Boolean functions

The boolean functions are used for checking, whether a certain name is actually a boolean, in which case it is treated differently, when printing and gathering channels. They also allow for properly printing boolean names.

3.2 Channel functions

The channel functions are used for manipulating names, which are to be printed as channels. The names need to be changed as they initially contain characters, which are not conventionally used in the Haskell language and might lead to conflicts.

3.3 Parameter functions

The parameter functions gather all of the names in a given process or processes. They are mostly used for gathering channel names when spawning channels or spawning processes and giving them the appropriate parameters. They also print errors, when the process syntax doesn't conform decomposed HO processes.

3.4 Process spawning functions

The process spawning functions are used solely when printing the `spawnLocal` function with the specified process and its arguments. They also print errors, when the process syntax doesn't conform decomposed HO processes.

3.5 Master process functions

The master process functions are used for printing the master process, which initializes the entirety of the decomposed HO process, as well as spawns all of the propagators, which are not enclosed in abstractions. The Cloud Haskell translation starts with theses functions and continues as the `haskellMaster` function calls the `haskellTrios` functions, which print the remaining decomposed process code.

3.6 Trio functions

The trio functions print the majority of the Cloud Haskell code, which is in essence the decomposed HO process, without the utility code such as the `rtable`, `main` function, `master` function etc..

The `haskellTrios` function takes in the HO process as the `PProc` data type and searches for trio patterns, since the data type does not incorporate them explicitly. For this reason, the `haskellTrios` as well as other search functions, such as the `haskellGetTrioParams` function, have cases for different types of actions, which includes application (`PApp`), variable application (`PAppVar`), input (`PRecv`), output (`PSend`), etc.. Depending on the case, the `haskellTrios` function call the appropriate trio function. These functions are as follows:

- `degenStartTrio` - Prints a degenerate trio for starting a process. It consists of a single action, which is an empty output. As all other trios, it prints an output indicating completion, when it is done. This serves to give feedback to the user when executing the program.

- `degenEndTrio` - Prints a degenerate trio for ending a process. It consists of a single action, which is an empty input.

- `degenAppTrio` - (NOTE: Due to an issue with the MISTY translation, this function does not print the proper input for the abstraction!) Prints a degenerate application trio, which first receives the context and then calls an abstraction. Function first defines the proper names, by checking the input types for the abstraction, then prints the abstraction call with either a single input variable or a tuple of multiple variables. It then prints the declaration of the abstraction. It receives the name and declaration of the abstraction by calling an external abstraction function.

- `degenAppVarTrio` - Works similarly to the `degenAppTrio` function, except in this case, it first receives a variable, which is the closed abstraction, then decodes it by collecting its input types, printing its type declaration, then printing the appropriate functions for decoding. Finally it applies the decoded abstraction with its input variables. The name and declaration are not used here, as the abstraction declaration is printed, when it is closed as sent, and the name doesn't matter, since it is received as a variable.

- `sendTrio` - Prints an output trio, which first receives the context, then it outputs either a set of variables (part of the context) or an abstraction closure. The function identifies, what to output by looking at the data type of the variable to be sent.

  If the variable is an abstraction, then the function calls external closure functions to obtain the abstraction name and definition. It prints a call of the closure function as well as the encoding function, which turn the abstraction into a `ByteString`, which is sent over. The abstraction is also provided with channels that are to be closed with it as parameters.

  If the variable is part of the context, then a call of the `ctGet` function is printed, which takes in the current context and the specified list of names to be sent and returns the modified context. In this case, the final propagator makes a call to the `ctSub` function, which returns the remainder of the context to be sent to the next trio.

- `recvTrio` - Prints an input trio, which first receives the context through the initial propagator, then receives an abstraction in the form of a `ByteString` and then adds it to the context with its appropriate variable name and sends it forward.

  This function also serve for printing control trios, since they have the same pattern of data types as the regular input trios.

3.7 Type definition functions

The type definition functions are used to retrieve the necessary type information from the decomposed process session types. The types are used primarily to identify the type of input for abstractions, when they are decoded, as their Haskell type has to be manually defined.

3.8 Abstraction functions

The abstraction functions print abstraction definitions. The primary part is the `haskellAbstraction` function, which takes in the processes executed by the abstraction as well as its name and input channels. It separates the channels in the abstraction itself, which are received as input, and those, which the abstraction must initialize itself (this is primarily propagators). Once it has obtained the two lists of channels, it puts the first as the abstraction parameters and spawns the second. It then spawns the inner processes, same as in restrictions and prints their trios. Finally, it prints the closure definition, but only, if the abstraction needs to be closed. It returns the entire printed abstraction as a string.

The `haskellTrios` functions serve primarily in identifying, if the input from the trio function is an abstraction or just a variable. They also define the name of the abstraction as well as its initialization.

3.9 Miscellaneous functions

These functions print out all of the utility functions outside of the simulated process as well as the imports. This includes the `rtable`, which is a required Cloud Haskell definition, when using closures and serialization, the context definition and function, and the main function.

4 Examples

The following examples are translated into working Cloud Haskell programs (apart from the deliberately wrong example). They are written in the MISTY input language, which is a Haskell version of

HO. The documentation for MISTY can be accessed at [`https://gitlab.com/aalen9/misty`]. Aside from basic functionality, they test for the following attributes:

- Basic abstraction sending (NamePassing, SendingAbstractions)
- Polyadic communication (MultipleInputs)
- Context passing (MultipleInputs)
- Environment channels, embedded in abstractions (NamePassing)
- Booleans (MultipleInputs, BoolAbstraction)
- Restrictions (NamePassing, MultipleInputs, BoolAbstraction)
- Polyadic input abstractions (MultipleInputs)
- Wrong (irreducible) processes (SimpleExample, AccumulatingChannels)

4.1 SimpleExample [Courtesy of A. Arslanagic]

A very simple HO process. This example shows the very basic HO process that can be coded in MISTY's input language. It includes the process `p`, which first waits to receive a channel and then sends it. Below is the channel `(Ch "u")`, which is how environment variables (channels that are not restrictions) are defined. Following is the typing for each channel. Then there is the actual environment definition, which is of the form `[(ChannelN, ST)]`, that is, a list of channels with their session types. Finally, there is the program definition, which includes the process, the environment and the name of the example. This example does not fully execute as the HO language is not reducible here.

```
module Examples.SimpleExample where

import Misty

p :: ChannelN -> ProcF ()
p u = do
        x <- receive u typx
        sendvar u x

-- channel
chu = Ch "u"

-- types
typx = typthunk
typu = typx :?> typx :!> STEnd
```

```
typthunk = (:>) STEnd
typbool = typthunk :!> STEnd

-- environment
env = [(Ch "u", typu)]

-- packed program : (process, environment, name)
program = p chu
name = "SimpleExample"
prog = (program, env, name)
```

4.2 SendingAbstractions

This example includes abstraction (`abstr`) definitions, which consists of a session type (`ST`) and an abstracted process (`ChannelN -> ProcF ()`) definition in the form of a lambda expression. It also shows that abstraction can be nested. In this case, abstraction `v` contains and sends the abstraction `w`.

```
module Examples.SendingAbstractions where

import Misty

r x = do
  send x v
  y <- receive (ChCmpl "u") typeY
  appvar y (Ch "True")
  end
  where
    typeZ = STEnd
    typeY = STBool
    w = abstr STEnd $ \b -> end
    v = abstr typeZ $ \z -> do
          send z w
          end

q x = do
  y <- receive x typx
  appvar y (Ch "u")
  end
  where
    typm = ((:>) ((:>) typbool :?> STEnd)) :!> STEnd
    typx = (:>) typm
```

```
    typeY = (:>) (((:>) STEnd) :!> STEnd)


p :: ProcF ()
p = do
  x <- new typeX
  par [q x, r (compl x)]

{-
typeB' =
typeW' = (:>) typeB'
typeZ' = typeW' :!> STEnd
typeV = (:>) typeZ'
-}
typthunk = (:>) STEnd
typbool = typthunk :!> STEnd  -- it must be of length 1
typeBOOL = STEnd
typeU = (:!>) ( (:>) STEnd ) STEnd
typeX = (:!>) ( (:>)  ( (:!>) ( (:>) STEnd ) STEnd ) ) STEnd

env = [((Ch "u"), typeU), ((Ch "True"), typeBOOL)]

name = "sendingAbstractions"
prog :: (ProcF (), [(ChannelN, ST)], String)
prog = (p, env, name)
```

4.3 BoolAbstraction

This example creates an abstraction, which is closed, sent over a channel and applied a boolean as its input. It shows how booleans are implemented in the MISTY input language, even though they are not originally part of HO. This is to give some function to the formal language of HO apart from describing pure communication. The abstraction `v` does not perform any function apart from printing the input boolean in the terminal. This example executes fully.

```
module Examples.BoolAbstraction where

import Misty

p :: ProcF ()
p = do
  send (Ch "x") v
```

```
  end
  where
    v = abstr STEnd $ \b -> end

q = do
  y <- receive (compl (Ch "x")) typeV
  appvar y (Ch "True")
  end
  where
    typeV = STBool

r = do
  par [p,q]
  end

env = [((Ch "x"),(:!>) (STBool) STEnd),((Ch "True"), STEnd)]

name = "boolAbstraction"
prog :: (ProcF (), [(ChannelN, ST)], String)
prog = (r, env, name)
```

4.4 MultipleInputs

This example creates multiple abstractions, which accumulate in the context and are later used one by one. This example showcases context passing, as the abstractions accumulate in process q, which repeatedly receives 3 abstractions in a row. It also showcases polyadic application (pappvar) for abstractions, as each abstraction has a different number of inputs. This example executes fully.

```
module Examples.MultipleInputs where

import Misty

r x y z = do
  send x $ pabstr [STEnd, STEnd, STEnd] (\[x1,y1,z1] -> end)
  send y $ pabstr [STEnd, STEnd] (\[x2,y2] -> end)
  send z $ abstr STEnd (\x3 -> end)

q x y z = do
  f1 <- receive x ((:>>) [STEnd, STEnd, STEnd])
  --[!<0->x>;0,!<0->x>;0,!<0->x>;0->x]).x [ax1,bx1,ax1]
  f2 <- receive y ((:>>) [STEnd, STEnd])
  f3 <- receive z ((:>) STEnd)
```

```
  par [(pappvar f1 [(Ch "True"), (Ch "False"), (Ch "True")]),(pappvar f2 [(Ch "
    False"), (Ch "True")]),(appvar f3 (Ch "False"))]

p :: ProcF ()
p = do
  par [q (Ch "m") (Ch "m") (Ch "m"), r (compl (Ch "m")) (compl (Ch "m")) (compl (Ch
     "m"))]

env = [(Ch "True", (STEnd)), (Ch "False", (STEnd)), (Ch "m", STEnd)]

name = "multipleInputs"
prog :: (ProcF (), [(ChannelN, ST)], String)
prog = (p, env, name)
```

4.5 NamePassing [Courtesy of A. Arslanagic]

A process that simulates name passing in HO, by encoding them into abstractions. The example was presented in the original HO decomposition paper from which MISTY stems and serves to show that HO can encode passing names, even though its original syntax only allows for passing abstractions.

```
module Examples.NamePassing where

import Misty

-- programs
q :: ChannelN -> ProcF ()
q u = do
        send u v
        y <- receive (ChCmpl "m") typy
        s <- new typs
        par [appvar y s, send (compl s) absb]
        where
          typm = ((:>) ((:>) typbool :?> STEnd)) :!> STEnd
          typx = (:>) typm
          typz = typx :?> STEnd
          typs = typz
          absb = abstr STEnd $ \b -> end
          typy = ((:>) ((:>) typbool :?> STEnd))
          v = abstr typz $ \z -> do
                                    x <- receive z typx
                                    appvar x (Ch "m")
r :: ChannelN -> ProcF ()
```

```
r u = do
        y <- receive u typy
        s <- new typs
        par [appvar y s, send (compl s) w]
        where
          typx' = (:>) typbool
          typz = typx' :?> STEnd
          typw' = (:>) typz
          typx = typw' :!> STEnd
          typm = ((:>) ((:>) typbool :?> STEnd)) :!> STEnd  -- = typx
          typw = (:>) typx
          typs = typw :?> STEnd
          typy = (:>) typs
          w' = abstr typz $ \z -> do
                                    x <- receive z STBool
                                    appvar x (Ch "True")
          w = abstr typx $ \x -> send x w'

p :: ProcF ()
p = do
      u <- new typu
      par [q u, r (compl u)]

-- types
typthunk = (:>) STEnd
typbool = typthunk :!> STEnd  -- it must be of length 1
typm = ((:>) ((:>) typbool :?> STEnd)) :!> STEnd
tx = (:>) typm
tyz = tx :?> STEnd
typv = (:>) tyz
typu = typv :!> STEnd

env = [(Ch "True", typbool), (Ch "m", typm)]

-- packed program : (process, environment, name)
name = "NamePassing"
prog :: (ProcF (), [(ChannelN, ST)], String)
prog = (p, env, name)
```

4.6 AccumulatingChannels

This example is an invalid HO process, which contains an abstraction, which takes in multiple channels and does not use them. This example showcases, why it is important to write correct and reducible HO processes. It fully executes, however, it treats all three input channels for the abstraction v as booleans, as they are not used. This is due to the way input channels for abstractions are translated in the Cloud Haskell extension for MISTY.

```haskell
module Examples.AccumulatingChannels where

import Misty


-- Translates properly, Executes, but doesn't give the proper result, since the process isn't valid.


p :: ProcF ()
p = do
        x <- receive (compl chu) ((:>>)  [STEnd,(:!>) (STBool) STEnd,(:!>) (STBool) STEnd])
        pappvar x [(Ch "True"),(Ch "y"),(Ch "z")]
        end

q :: ProcF ()
q = do
      send chu v
      end
      where
        v = pabstr [(:!>) (STBool) STEnd, STEnd,(:!>) (STBool) STEnd] (\[x,y,z] -> end)

-- channel
chu = Ch "u"

-- environment
env = [(Ch "u", STEnd), (Ch "True", STEnd), (Ch "y", STEnd), (Ch "z", STEnd)]

program = par [p,q]
-- packed program : (process, environment, name)
name = "AccumulatingChannels"
prog = (program, env, name)
```