Advanced Programming Seminar 5

Mart Lubbers

November 13, 2018

Assignment 5 recap

Assignment 5 recap

Simple tasks

```
enterStudent :: Task Student
enterStudent = enterInformation "Enter a student" []

enterStudentList :: Task [Student]
enterStudentList = enterInformation "Enter a student" []

updateStudent :: Student → Task Student
updateStudent s = updateInformation "Update a student" [] s

favouriteStudent :: [Student] → Task Student
favouriteStudent sl = enterChoice "Pick a student" [] sl
```

Intermezzo: Record Selection

- The compiler has to know the type of the record.

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {field = not t.field}
```

Intermezzo: Record Selection

- The compiler has to know the type of the record.
- Moreover, it needs to know the record a field selector belongs to.

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {field = not t.field}
```

Intermezzo: Record Selection

- The compiler has to know the type of the record.
- Moreover, it needs to know the record a field selector belongs to.
- However, the function type is NOT used to determine this.

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {field = not t.field}
```

Intermezzo: Record Selection

- The compiler has to know the type of the record.
- Moreover, it needs to know the record a field selector belongs to.
- However, the function type is NOT used to determine this.

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {field = not t.field}
```

Intermezzo: Record Selection

- The compiler has to know the type of the record.
- Moreover, it needs to know the record a field selector belongs to.
- However, the function type is NOT used to determine this.

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {field = not t.field}
```

```
Error [...]: could not determine the type of this
record
```

Intermezzo: Record Selection

Let's explicitly tell the compiler the type of the record:

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {T1 | field = not t.field}
```

Intermezzo: Record Selection

Let's explicitly tell the compiler the type of the record:

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {T1 | field = not t.field}
```

```
Error [...]:  field ambiguous selector specified
```

Intermezzo: Record Selection

Let's explicitly tell the compiler the type of the record:

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {T1 | field = not t.field}
```

```
Error [...]: field ambiguous selector specified
```

Works also in a pattern match:

```
neg {T1|field} = {T1 | field = not field}
```

Intermezzo: Record Selection

Let's explicitly tell the compiler the type of the record AND the record the field belongs to:

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {T1 | field = not t.T1.field}
```

Intermezzo: Record Selection

Let's explicitly tell the compiler the type of the record AND the record the field belongs to:

```
:: T1 = {field :: Bool}
:: T2 = {field :: Bool}

neg :: T1 → T1
neg t = {T1 | field = not t.T1.field}
```

WIN

Assignment 5 recap

Modifying the editors

```
selectStudentOnlyName :: [Student] → Task Student
selectStudentOnlyName sl = enterChoice "Pick a student"
    [ChooseFromDropdown λs→s.Student.name] sl

selectStudentFormat :: [Student] → Task Student
selectStudentFormat sl = enterChoice "Pick a student"
    [ChooseFromDropdown gToString{|*|}] sl

selectPartner :: [Student] → Task [Student]
selectPartner sl = enterMultipleChoice "Pick a partner"
    [ChooseFromCheckGroup
       λs→s.Student.name + "(" + gToString{|*|} s.Student.bama + ")"] sl
```

Assignment 5 recap

Modifying the editors

```
selectStudentOnlyName :: [Student] → Task Student
selectStudentOnlyName sl = enterChoice "Pick a student"
    [ChooseFromDropdown λs→s.Student.name] sl

selectStudentFormat :: [Student] → Task Student
selectStudentFormat sl = enterChoice "Pick a student"
    [ChooseFromDropdown gToString{|*|}] sl

selectPartner :: [Student] → Task [Student]
selectPartner sl = enterMultipleChoice "Pick a partner"
    [ChooseFromCheckGroup
       λs→s.Student.name + "(" + gToString{|*|} s.Student.bama + ")"] sl
```

Assignment 5 recap

Modifying the editors

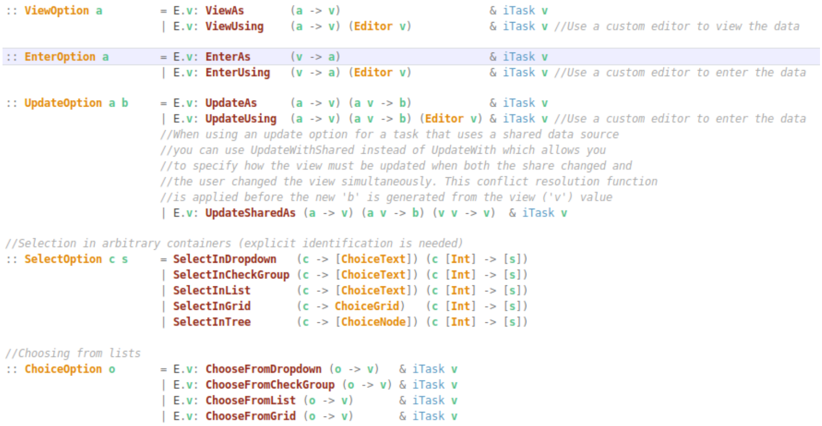
There are many ways of modifying the editors. To find them all, see `iTasks/WF/Tasks/Interaction.dcl`[1]

[1]Or browse it live at `https://cloogle.org/src/#iTasks/iTasks/WF/Tasks/Interaction`

Assignment 5 recap

Modifying the editors

There are many ways of modifying the editors. To find them all, see `iTasks/WF/Tasks/Interaction.dcl`[1]



```
:: ViewOption a         = E.v: ViewAs      (a -> v)                      & iTask v
                        | E.v: ViewUsing   (a -> v) (Editor v)           & iTask v //Use a custom editor to view the data

:: EnterOption a        = E.v: EnterAs     (v -> a)                      & iTask v
                        | E.v: EnterUsing  (v -> a) (Editor v)           & iTask v //Use a custom editor to enter the data

:: UpdateOption a b     = E.v: UpdateAs    (a -> v) (a v -> b)           & iTask v
                        | E.v: UpdateUsing (a -> v) (a v -> b) (Editor v) & iTask v //Use a custom editor to enter the data
                        //When using an update option for a task that uses a shared data source
                        //you can use UpdateWithShared instead of UpdateWith which allows you
                        //to specify how the view must be updated when both the share changed and
                        //the user changed the view simultaneously. This conflict resolution function
                        //is applied before the new 'b' is generated from the view ('v') value
                        | E.v: UpdateSharedAs (a -> v) (a v -> b) (v v -> v)  & iTask v

//Selection in arbitrary containers (explicit identification is needed)
:: SelectOption c s     = SelectInDropdown   (c -> [ChoiceText]) (c [Int] -> [s])
                        | SelectInCheckGroup (c -> [ChoiceText]) (c [Int] -> [s])
                        | SelectInList       (c -> [ChoiceText]) (c [Int] -> [s])
                        | SelectInGrid       (c -> ChoiceGrid)   (c [Int] -> [s])
                        | SelectInTree       (c -> [ChoiceNode]) (c [Int] -> [s])

//Choosing from lists
:: ChoiceOption o       = E.v: ChooseFromDropdown (o -> v)   & iTask v
                        | E.v: ChooseFromCheckGroup (o -> v) & iTask v
                        | E.v: ChooseFromList (o -> v)       & iTask v
                        | E.v: ChooseFromGrid (o -> v)       & iTask v
```

[1]Or browse it live at
`https://cloogle.org/src/#iTasks/iTasks/WF/Tasks/Interaction`

Generic printing

```
generic gToString a :: a → String
```

Some people were smart. . .

Generic printing

```
generic gToString a :: a → String
```

Some people were smart. . .

```
generic gToString a :: a → String
gToString{|BaMa|} Bachelor = "Bachelor"
gToString{|BaMa|} Master = "Master"
gToString{|Student|} ...
```

Generic printing

```
generic gToString a :: a → String
```

Some people were smart...

```
generic gToString a :: a → String
gToString{|BaMa|} Bachelor = "Bachelor"
gToString{|BaMa|} Master = "Master"
gToString{|Student|} ...
```

But a real implementation is almost trivial:

Generic printing

```
generic gToString a :: a → String
```

Some people were smart. . .

```
generic gToString a :: a → String
gToString{|BaMa|} Bachelor = "Bachelor"
gToString{|BaMa|} Master = "Master"
gToString{|Student|} ...
```

But a real implementation is almost trivial:

```
gToString{|Int|} i = toString i
gToString{|String|} s = s
gToString{|UNIT|} _ = ""
gToString{|RECORD|} fx (RECORD x) = "{" + fx x + "}"
gToString{|FIELD of {gfd_name}|} fx (FIELD x) = gfd_name + "=" + fx x + " "
gToString{|PAIR|} fx fy (PAIR x y) = fx x + fy y
gToString{|EITHER|} fx fy (LEFT x) = fx x
gToString{|EITHER|} fx fy (RIGHT y) = fy y
gToString{|CONS of {gcd_name}|} fx (CONS x) = gcd_name + fx x
gToString{|OBJECT|} fx (OBJECT x) = fx x
```

Assignment 5 recap

Update a single field using parallel combinators

This is a sneak preview for the next assignment:

Assignment 5 recap

Update a single field using parallel combinators

This is a sneak preview for the next assignment:

```
changeName :: Student → Task Student
changeName s
    =  viewInformation "Student to change" [] s
    ||- updateInformation "New name" [updater] s
where
    updater = UpdateAs (λs→s.Student.name) (λs n→{Student | s & name=n})
```

Assignment 5 recap

Update a single field using editor combinators

```
changeNameEdcomb :: Student → Task Student
changeNameEdcomb s
    = updateInformation "New name" [UpdateUsing id (λ_ v→v) nameEditor] s
where
    nameEditor :: Editor Student
    nameEditor = bijectEditorValue
        (λ{name=n,snum=s,bama=b,year=y}→(n, s, b, y))
        (λ(n,s,b,y)→{name=n,snum=s,bama=b,year=y})
        (container4
            (gEditor{|*|}≪@labelAttr "name")
            (withChangedEditMode toView gEditor{|*|}≪@labelAttr "snum")
            (withChangedEditMode toView gEditor{|*|}≪@labelAttr "bama")
            (withChangedEditMode toView gEditor{|*|}≪@labelAttr "year")
        )

    toView (Update a) = View a
    toView v = v

bijectEditorValue :: !(a → b) !(b → a) !(Editor b) → Editor a
```

Assignment 5 recap

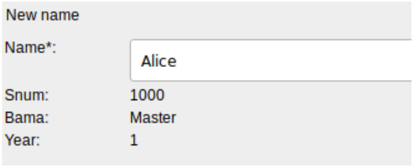
Update a single field using editor combinators

You can totally customize your editors using these functions.

Assignment 5 recap

Update a single field using editor combinators

You can totally customize your editors using these functions.



New name

Name*:	Alice
Snum:	1000
Bama:	Master
Year:	1

Assignment 5: iTasks Combinators

The types reveal the semantics:

Parallel combinators

```
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iTask a & iTask b
(-||-)  infixr 3 :: (Task a) (Task a) → Task a     | iTask a
(||-)   infixr 3 :: (Task a) (Task b) → Task b     | iTask a & iTask b
(-||)   infixl 3 :: (Task a) (Task b) → Task a     | iTask a & iTask b

anyTask  :: [Task a] → Task a   | iTask a
allTasks :: [Task a] → Task [a] | iTask a
```

Assignment 5: iTasks Combinators

The types reveal the semantics:

Parallel combinators

```
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b) | iTask a & iTask b
(-||-)  infixr 3 :: (Task a) (Task a) → Task a     | iTask a
(||-)   infixr 3 :: (Task a) (Task b) → Task b     | iTask a & iTask b
(-||)   infixl 3 :: (Task a) (Task b) → Task a     | iTask a & iTask b

anyTask  :: [Task a] → Task a   | iTask a
allTasks :: [Task a] → Task [a] | iTask a
```

Sequential combinators

```
(>>~) infixl 1 :: (Task a) (a → Task b) → Task b | iTask a & iTask b
(>>-) infixl 1 :: (Task a) (a → Task b) → Task b | iTask a & iTask b
(>>=) infixl 1 :: (Task a) (a → Task b) → Task b | iTask a & iTask b
(>>|) infixl 1 :: (Task a) (     Task b) → Task b | iTask a & iTask b
```

Assignment 5: iTasks Combinators

Step combinator

```
(>>*) infixl 1 :: (Task a) [TaskCont a (Task b)] → Task b | iTask a & iTask
    b
:: TaskCont a b
    =      OnValue ((TaskValue a) → Maybe b)
    |      OnAction Action ((TaskValue a) → Maybe b)
    | ∃e: OnException (e → b) & iTask e
    |      OnAllExceptions (String → b)
:: Action = Action String
```

Intermezzo: Task values

$$\begin{tikzcd}
\mathit{NoValue} \arrow[r, shift left] \arrow[rr, bend left] & \mathit{Unstable} \arrow[l, shift left] \arrow[r] & \mathit{Stable}
\end{tikzcd}$$

Assignment 5: iTasks Combinators

Step helpers

```
always      :: b                          (TaskValue a) → Maybe b
never       :: b                          (TaskValue a) → Maybe b
hasValue    :: (a → b)                    (TaskValue a) → Maybe b
ifStable    :: (a → b)                    (TaskValue a) → Maybe b
ifUnstable  :: (a → b)                    (TaskValue a) → Maybe b
ifValue     :: (a → Bool)        (a → b)  (TaskValue a) → Maybe b
ifCond      :: Bool b                     (TaskValue a) → Maybe b
withoutValue :: (Maybe b)       (TaskValue a) → Maybe b
withValue   :: (a → Maybe b) (TaskValue a) → Maybe b
withStable  :: (a → Maybe b) (TaskValue a) → Maybe b
withUnstable :: (a → Maybe b) (TaskValue a) → Maybe b
```

Assignment 5: iTasks Combinators

Derivations

```
(>>=) lhs rhs = lhs >>*
    [ OnValue (ifStable rhs)
    , OnAction (Action "Continue") (hasValue rhs)
    ]
(>>-) lhs rhs = lhs >>* [OnValue (ifStable rhs)]
(>>~) lhs rhs = lhs >>* [OnValue (hasValue rhs)]
(>>|) lhs rhs = lhs >>= λ_ → rhs

sequence [] = return []
sequence [t:ts] = t >>= λtv → sequence tv >>= λtvs → return [tv:tvs]
```

Assignment 5: iTasks Combinators

Examples of step

```
palindrome :: Task (Maybe String)
palindrome
    =   enterInformation "Enter a palindrome" []
    >>*
        [OnAction (Action "Ok")
            (ifValue isPalindrome (return o Just))
        ,OnAction (Action "Cancel")
            (always (return Nothing))
        ]
```

Assignment 5: iTasks Combinators

Examples of step

```
palindrome :: Task (Maybe String)
palindrome
    =   enterInformation "Enter a palindrome" []
    >>*
        [OnAction (Action "Ok")
            (ifValue isPalindrome (return o Just))
        ,OnAction (Action "Cancel")
            (always (return Nothing))
        ]
```

demo

Assignment 5: iTasks Combinators

Transforming the task value

```
(@)  infixl 1 :: (Task a) (a → b) → Task b
(@?) infixl 1 :: (Task a) ((TaskValue a) → TaskValue b) → Task b
(@!) infixl 1 :: (Task a) b → Task b
```

Assignment 5: iTasks Combinators

Shared Data Sources

- Atomic read write and update operations
- Communication between tasks
- Some shares are persistent between executions

Assignment 5: iTasks Combinators

Shared Data Sources

- Atomic read write and update operations
- Communication between tasks
- Some shares are persistent between executions

```
:: SDS p r w = ...
:: Shared a :== SDS () a a
:: ReadWriteShared r w :== SDS () r w

get   ::          (ReadWriteShared a w) → Task a | iTask a
set   :: a        (ReadWriteShared r a) → Task a | iTask a & TC r
upd   :: (r → w) (ReadWriteShared r w) → Task w | iTask r & iTask w
watch ::          (ReadWriteShared r w) → Task r | iTask r
```

Assignment 5: iTasks Combinators

Create Shares

Named shares

```
sharedStore :: String a → Shared a | iTask a
```

Anonymous shares

```
withShared :: !b !((Shared b) → Task a) → Task a | iTask a & iTask b

editList :: Task [Int]
editList = withShared [] λshare→
       viewSharedInformation "Share" [] share
    -||- forever (enterInformation "New Item" [] >>=λel→upd (λl→[el:l])
     share))
```

Good Luck

Demo?