THE UNIVERSITY *of* EDINBURGH
informatics

Inf 2D – Coursework 1

Haskell Refresher Lecture

Petros Papapanagiotou, Jiawei Zheng
pe.p@ed.ac.uk, jw.zheng@ed.ac.uk
Feb 2022

Haskell

- Purely functional! : *"Everything is a function!"*

- Main topics:
  - Recursion
  - Currying
  - Higher-order functions
  - List Processing functions such as map, filter, foldl, sortBy, etc.
  - The Maybe monad

- For more: *http://www.haskell.org/haskellwiki/Haskell*

Haskell refresher!

A CSP

- Consists of:
  - **X** : a set of **Variables**
  - **D** : a set of **Domains**
  - **C** : a set of **Constraints**

X : a set of **Variables**

- Variables as `String`s:

```
type Var = String
```

eg. `"X"`

D : a set of **Domains**

- Domain as a list of allowable (`Int`) values for each variable:

```
type Domain = [Int]
```

eg. `[1,2,3]`

- D as a list of `Domain`s for each `Var`:

```
type Domains = [(Var,Domain)]
```

eg. `[("X",[1,2,3]),("Y",[1,2])]`

State of a CSP: Assignment

- *"Assignment of values to some or all of the variables"*
  - R&N §6.1 / NIE Ch.7 §1

- Assignment as custom type:

```
newtype AssignedVar = ...
```

eg. `x=1`

```
type Assignment = [AssignedVar]
```

eg. `[x=1,y=2]`

- Manipulate it using functions.

Assignment functions

- `assign ::`

  `(Var,Int) -> Assignment -> Assignment`

  - **eg.** `assign (“x”,1) []` → `[x=1]`
  - **eg.** `assign (“y”,2) [x=1]` → `[y=2,x=1]`
  - **eg.** `assign (“x”,2) [x=1]` → `[x=2]` (Updated!)

Assignment functions

- `assign ::`

  `(Var,Int) -> Assignment -> Assignment`

  - **eg.** `assign (“x”,1) []` → `[x=1]`
  - **eg.** `assign (“y”,2) [x=1]` → `[y=2,x=1]`
  - **eg.** `assign (“x”,2) [x=1]` → `[x=2]` (Updated!)

- **Care!!**: `[y=2,x=1]` is really:

  `assign (“y”,2) (assign (“x”,1) [])`

Assignment functions

- `lookupVar ::`

  `Assignment -> Var -> Maybe Int`

  - **eg.** `lookupVar [x=2] "x" → Just 2`
  - **eg.** `lookupVar [x=2] "y" → Nothing`

- `isAssigned ::`

  `Assignment -> Var -> Bool`

  - **eg.** `isAssigned [x=2] "x" → True`
  - **eg.** `isAssigned [x=1] "y" → False`

Relations

- *"Give me a scope and a state and I'll tell you if it's ok!"*
  – Relation

```
type Relation =
[Var] -> Assignment -> Bool
```

Relations

- **Example:** `varsDiff :: Relation`
  - Ensures two variables are different.
  - Scope = *"which variables?"*
  - Assignment = *"what state should I check?"*

  - Examples:
    - `varsDiff ["x","y"] [x=1,y=2]` → `True`
    - `varsDiff ["x","y"] [x=1,y=1]` → `False`
  - Care for unassigned variables!!
    - `varsDiff ["x","y"] [x=1]` → `True` (!!)
    - `varsDiff ["x","y"] []` → `True` (!!)

Constraint functions

- `checkConstraint :: Constraint -> Assignment -> Bool`
- `checkConstraints :: [Constraint] -> Assignment -> Bool`
- `scope :: Constraint -> [Var]`
- `isConstrained :: Var -> Constraint -> Bool`
- `neighboursOf :: Var -> Constraint -> [Var]`

Constraint constructors

- Functions (wrappers) that construct a `Constraint` from a `Relation`.
- Already done for you!!

- eg.

```
varsDiffConstraint ::
Var -> Var -> Constraint
```

- `varsDiffConstraint "x" "y"` → *`Constraint`*

CSP

- *Bringing it all together...*

- CSPs as a custom type:

```
newtype CSP = ...
```

- Constructor:

```
CSP ( String , Domains , [Constraint] )
```

The BACKTRACK algorithm

```
bt :: CSP -> Maybe Assignment
bt csp = btRecursion csp []

btRecursion :: CSP -> Assignment -> Maybe Assignment
btRecursion csp assignment =
    if (isComplete csp assignment) then Just assignment
    else findConsistentValue $ getDomain var csp
      where var = firstUnassignedVar assignment csp
            findConsistentValue vals =
              case vals of  -- recursion over the possible values
                            -- instead of for-each loop
                []     -> Nothing
                val:vs ->
                  if (isConsistentValue csp assignment (var,val))
                  then if (isNothing result)
                       then ret
                       else result
                  else ret
                     where result = btRecursion csp $ assign (var,val) assignment
                           ret = findConsistentValue vs
```

The BACKTRACK algorithm

```
bt :: CSP -> (Maybe Assignment, Int)
bt csp = btRecursion csp []

btRecursion :: CSP -> Assignment -> (Maybe Assignment, Int)
btRecursion csp assignment =
    if (isComplete csp assignment) then (Just assignment,0)
    else findConsistentValue $ getDomain var csp
      where var = firstUnassignedVar assignment csp
            findConsistentValue vals =
              case vals of  -- recursion over the possible values
                            -- instead of for-each loop
                []      -> (Nothing,0)
                val:vs ->
                  if (isConsistentValue csp assignment (var,val))
                  then if (isNothing result)
                       then (ret,nodes+nodes'+1)
                       else (result,nodes+1)
                  else (ret,nodes'+1)
                     where (result,nodes) = btRecursion csp $
                                                    assign (var,val) assignment
                           (ret,nodes') = findConsistentValue vs
```

Unit Testing!