GenI: Natural language generation in Haskell

Eric Kow (kowey)

INRIA/LORIA/UHP

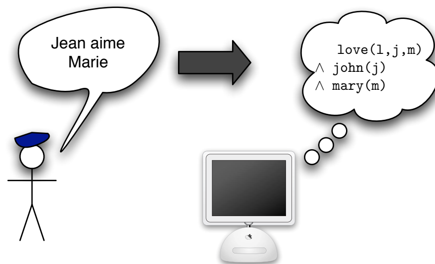
17 September 2006
Portland Oregon

Outline of this talk

1. Natural Language Generation and GenI
2. Haskell
   - Typeclasses
   - Monads
3. Problems faced
   - Timeouts
   - Profiling

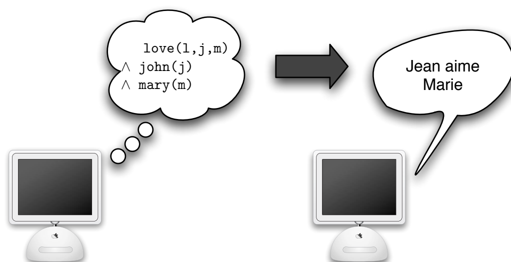
Natural Language Processing

- Natural Language Processing (NLP) - automatic processing of *human* language
- Examples: Speech recognition, document summarisation, machine translation, etc
- Common theme: translate human language to abstract representation



Natural Language Generation

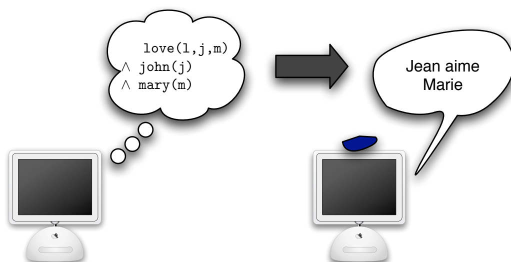
- Natural Language Generation - an NLP task
- Translate abstract representation to human language



- Known to be NP-complete (polynomial time in practice?)
- Finding efficient algorithms important - applications need to be reactive

Natural Language Generation

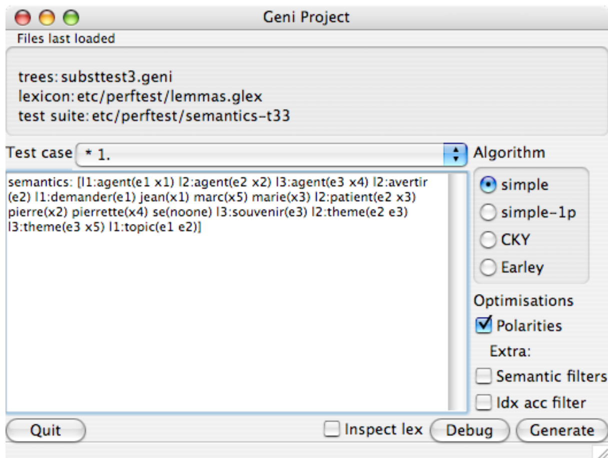
- Natural Language Generation - an NLP task
- Translate abstract representation to human language



- Known to be NP-complete (polynomial time in practice?)
- Finding efficient algorithms important - applications need to be reactive

GenI

Platform for experimenting with generation algorithms and optimisations



GenI

Complicated! (14 000 lines literate Haskell; 8000 without comments)

1. 2 core algorithms - different inputs, internal representations
2. Sets of optimisations - dependencies on each other, on algorithms
3. Graphical debugger for each algorithm

Complex type hierarchy

A sample of GenI types

```
data GeniVal = GConst String | GVar String

data FirstOrderTerm = (String, [GeniVal])

type AttValPair = (String, GeniVal)
data GNode    = GNode [AttValPair] [AttValPair]
```

Task: unification

- Unification between two: `GeniVal`, `[GeniVal]` and `[AttValPair]`
- Simple (not Prolog-term unification)
- But must propagate variable substitutions (e.g. $X \leftarrow foo$) throughout the type hiearchy

Task: unification

- Unification between two: `GeniVal`, `[GeniVal]` and `[AttValPair]`
- Simple (not Prolog-term unification)
- But must propagate variable substitutions (e.g. $X \leftarrow foo$) throughout the type hiearchy
- Solution: typeclass

Replaceable class

```
class Replaceable a where
 replace :: [(String,GeniVal)] -> a -> a
```

Replaceable

First implement `Replaceable` for a single `GeniVal`

Perform the replacements in order

```
instance Replaceable GeniVal where
 replace sl v = foldl replaceOne v sl
  where
   replaceOne (GVar x) (s1,s2) | x == s1 = s2
   replaceOne gx _ = gx
```

Then infer its implementation for anything containing a `GeniVal`...

Replaceable

2 for the price of 1

```
-- covers both (FirstOrderTerm and AttValPair)
instance Replaceable v => Replaceable (String,v) where
 replace s (a,v) = (a, replace s v)
```

Replaceable

2 for the price of 1

```
-- covers both (FirstOrderTerm and AttValPair)
instance Replaceable v => Replaceable (String,v) where
 replace s (a,v) = (a, replace s v)
```

Hooray for typeclasses!

```
instance Replaceable a => Replaceable [a] where
  replace s = map (replace s)
```

Task: "α-conversion"

- α-conversion-esque operation = renaming variables in a term (helps us avoid unification errors)
- Subtask: collect the variables in some datatype

Task: "α-conversion"

- α-conversion-esque operation = renaming variables in a term (helps us avoid unification errors)
- Subtask: collect the variables in some datatype
- Solution: typeclass

Collectable class

```
class Collectable a where
  collect :: a -> Set String -> Set String
```

Collectable

Same N-for-price-of-1 effect!

```
instance Collectable GeniVal where
 collect (GVar v) s = Set.insert v s
 collect _ s = s

instance Collectable v => Collectable (String, v) where
 collect (_,b) = collect b

instance Collectable a => Collectable [a] where
 collect l s = foldr collect s l
```

And "α-conversion"?

Define α-conversion as "add a unique suffix to all variables" (good enough for our needs).

Combine replace and collect

```
alphaConvert :: (Collectable a, Replaceable a)
             => String -> a -> a
alphaConvert suffix x =
  let vars  = Set.elems (collect x Set.empty)
      subst = map (\v -> (v, GVar (v ++ suffix))) vars
  in replace subst x
```

Can now do "α-conversion" on (almost) anything!

Task: output to Graphviz

GenI has a different graphical debugger for each algorithm.

- Need visual representation of trees, graphs, etc (GraphViz)
- Representation must be parameterisable
- But different datatypes require different parameters

Task: output to Graphviz

GenI has a different graphical debugger for each algorithm.

- Need visual representation of trees, graphs, etc (GraphViz)
- Representation must be parameterisable
- But different datatypes require different parameters
- Solution: *Multi-parameter* typeclasses

GvizShow class

```
class GvizShow flag i where
  gvizShow :: flag -> i -> String -> String
```

Graphical debuggers - Simple algorithm


```
i :: SimpleItem
flag :: Bool
```

Graphical debuggers - CKY algorithm

GenI Debugger – CKY edition

lexical selection | automata | CKY-session

72 g2 demander /demand
71 g2 demander /demand
69 g2.1 demander /demar
70 g3 N demander S /dem
___RESULTS___
>278 Pierre avertir Marie J
>277 Pierre avertir Jean de
>276 Pierre avertir Marie c
>275 Pierre avertir que Jea
>268 Jean demander Pierr
>265 Jean demander Pierr
>262 Jean demander Pierr
>259 Jean demander Pierr
>247 Pierre avertir Marie J
>241 Jean demander Pierr
>238 Jean demander Pierr
>237 Pierre avertir Jean de
>231 Jean demander Pierr
>228 Jean demander Pierr
>227 Pierre avertir Marie J
>226 Pierre avertir Marie c
>220 Jean demander Pierr
>217 Jean demander Pierr
>216 Pierre avertir Jean de
>215 Pierre avertir que Jea

278 s #
269 s
100 n:Pierre # 88 vp # 89 n:Marie # 53 s *
50 n ! 51 v:avertir # 52 n !

0 Pierre 2 avertir 4 Marie 6 Jean 8 demander

Show... full derivation src tree features
Deriviations (1) 1 Node 45 Go to node Pop back
itr 301 chart sz: 301
comparisons: 0
Start over Leap by... 301 step(s) Continue

```
i :: CkyItem
flag :: (Int,Bool,Bool)
```

Task: unpacking

- GenI works by building "chart items" - essentially trees
- For combinatory/efficiency reasons, each chart item packs a list of other chart items
- At end of processing, must "unpack" the results
- Solution: List monad

With monads

```
unpack :: Tree [a] -> [Tree a]
unpack (Node px pks) =
  do x  <- px
     ks <- mapM unpack pks
     return (Node x ks)
```

Without monads

```
unpack2 :: Tree [a] -> [Tree a]
unpack2 (Node px pks) =
 let next pk [] =
       map (\k -> [k]) (unpack2 pk)
     next pk ls =
       concatMap (\k -> map (k:) ls) (unpack2 pk)
 in case foldr next [] pks of
    [] -> map (\x -> Node x []) px
    ks -> concatMap (\x -> (map (Node x) ks)) px
```

Without monads

```
unpack2 :: Tree [a] -> [Tree a]
unpack2 (Node px pks) =
 let next pk [] =
       map (\k -> [k]) (unpack2 pk)
     next pk ls =
       concatMap (\k -> map (k:) ls) (unpack2 pk)
 in case foldr next [] pks of
    [] -> map (\x -> Node x []) px
    ks -> concatMap (\x -> (map (Node x) ks)) px
```

Beurk!

Without monads

```
unpack2 :: Tree [a] -> [Tree a]
unpack2 (Node px pks) =
 let next pk [] =
        map (\k -> [k]) (unpack2 pk)
     next pk ls =
        concatMap (\k -> map (k:) ls) (unpack2 pk)
 in case foldr next [] pks of
    [] -> map (\x -> Node x []) px
    ks -> concatMap (\x -> (map (Node x) ks)) px
```

Beurk! (French for "Yuck!")

Task: general book-keeping

- Solution: `State` monad
- Each algorithm shares a common "architecture"

Foo`Status` ,*Foo*`Item` and *Foo*`State`

```
data CkyStatus = CkyStatus -- details elided
data CkyItem   = CkyItem   -- details elided
type CkyState  = State CkyStatus
```

Task: general book-keeping

- Solution: `State` monad
- Each algorithm shares a common "architecture"

Foo`Status` ,*Foo*`Item` and *Foo*`State`

```
data CkyStatus = CkyStatus -- details elided
data CkyItem   = CkyItem   -- details elided
type CkyState  = State CkyStatus
```

- Inference rules take existing items and produce new items, updating the book-keeping information along the way

Inference rules

```
ckyRule1 :: CkyItem -> CkyState [CkyItem]
ckyRule2 :: CkyItem -> CkyState [CkyItem]
-- ...
```

Task: count things (cleanly)

- Must count things (e.g. chart items) to understand performance
- Solution: State transformer monad

Version 1 - no counter

```
type CkyState = State CkyStatus
ckyRuleX :: CkyItem -> CkyState [CkyItem]
```

Task: count things (cleanly)

- Must count things (e.g. chart items) to understand performance
- Solution: State transformer monad

Version 2 - one counter

```
type CkyState a = StateT CkyStatus (State Int) a
ckyRuleX :: CkyItem -> CkyState [CkyItem]

incrCounter :: StateT st (State Int) ()
incrCounter = lift $ modify (+1)
```

Task: count things (cleanly)

- Must count things (e.g. chart items) to understand performance
- Solution: State transformer monad

Version 3 - list of named counters

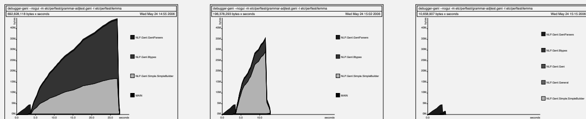
```
type Ctr = (String,Int)
type CkyState a = StateT CkyStatus (State [Ctr]) a
ckyRuleX :: CkyItem -> CkyState [CkyItem]

incrCounter :: String -> StateT st (State [Ctr]) ()
incrCounter s = lift $ modify (map helper)
 where helper (n,v) | n == s = (n,v+1)
       helper c = c
```

Problem - Timeouts

- Useful to time GenI out if it takes unreasonably long.
- We use Marlow 2000 asynchronous exception approach
- Gracefully recovering from timeout (in `State` monad) still tricky
- Wish : easy way to do timeouts and recover from them

Problem - Profiling

Before, during and after



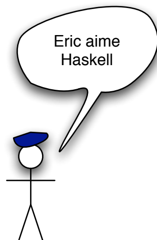
- Profiling was useful!
- But don't know how to use it effectively (improvements largely from staring at code)
- Wish : *Haskell Profiling for Dummies*
- Wish : Swiss Army knife of profiling (ghcprof + Blobs + Haskell Charts) == reduced learning curve?

Conclusion

- Lessons learned: Haskell keeps code tidy and tidy.
    1. Typeclasses keep code highly factorised.
    2. Monads keep code simple and modular.
- Open questions: Timeouts and profiling are tricky. Help?

Conclusion

- Lessons learned: Haskell keeps code tidy and tidy.
  1. Typeclasses keep code highly factorised.
  2. Monads keep code simple and modular.
- Open questions: Timeouts and profiling are tricky. Help?
- Overall:



Thanks!

`@karma+`

- wxHaskell
- QuickCheck
- Parsec
- HaXmL (DtdToHaskell)
- #haskell (irc.freenode.net)

Task: put chart items in right place

- GenI produces new chart items at each iteration.
- Different chart items have different roles to be determined from properties of item
- Complex rules == messy!

Task: put chart items in right place

- GenI produces new chart items at each iteration.
- Different chart items have different roles to be determined from properties of item
- Complex rules == messy!
- Solution: dispatch rule combinators

Dispatch filters

```
type DispatchFilter s a = a -> s (Maybe a)

(>-->) :: (Monad s) => DispatchFilter s a
                    -> DispatchFilter s a
                    -> DispatchFilter s a
f1 >--> f2 =
  \x -> f1 x >>= maybe (return Nothing) f2
```

Dispatch filters?

From previous slide

```
type DispatchFilter s a = a -> s (Maybe a)

(>-->) :: (Monad s) => DispatchFilter s a
                    -> DispatchFilter s a
                    -> DispatchFilter s a
f1 >--> f2 =
  \x -> f1 x >>= maybe (return Nothing) f2
```

- Each filter tries to put an item somewhere.
  - If item is trapped, return `Nothing` (our job is done)
  - If item passes through, return `Just` (perhaps modify the item)
- Use `>-->` combinator to sequence filters, and `condFilter` combinator to express filter `if then else`

Using dispatch filters

DispatchFilters in action

```
dispatchSim1 :: DispatchFilter SimStatus SimItem
dispatchSim1 =
 condFilter isResult1
   (dpTbFailure >--> dpRootCatFailure >--> dpToResults)
   (dpTreeLimit >--> dpAux >--> dpToAgenda)

dispatchSim2 :: DispatchFilter SimStatus SimItem
dispatchSim2 =
 condFilter isResult2
   (dpTbFailure >--> dpRootCatFailure >--> dpToResults)
   (dpTreeLimit >--> maybeDpIaf >--> dpToAgenda)
```

- Relationship between filters readily apparant
- Filters can be shared by different dispatch processes