agdARGS

Command Line Arguments, Options and Flags

Guillaume Allais

University of Strathclyde

Idris Developers Meeting, March 2015

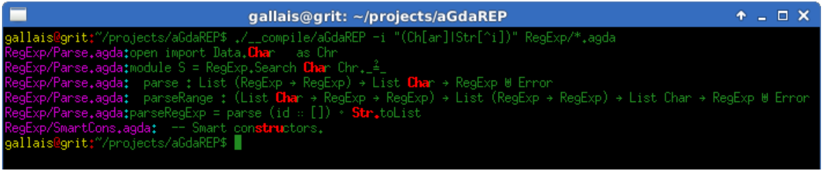
Crazy thought: let's compile some programs!

Bored of the stereotype that in Type Theory we never go anywhere past typechecking or producing `.tex` documents (which, by the way, Travis can help with! `http://blog.gallais.org/travis-builds`).

Decided to build a simple executable because I can.

https://github.com/gallais/aGdaREP

My simple pick: a certified regexp matcher leading to an implementation of grep.



Lots of fun implementing, optimizing and extending the correct by construction matcher (see Alexandre Agular and Bassel Mannaa's 2009 "Regular Expressions in Agda"),

But... then we need a user-facing interface!

A cheeky account of my journey

First step: add a binding to the important Haskell function...

```
module Bindings.Arguments.Primitive where

open import IO.Primitive
open import Data.List
open import Data.String

{-# IMPORT System.Environment #-}

postulate
  getArgs : IO (List String)

{-# COMPILED getArgs System.Environment.getArgs #-}
```

But wait! There's more!

Then lift the bound primitive to the IO type actually used at a high level of abstraction:

```
module Bindings.Arguments where

open import Data.List
open import Data.String
open import IO
import Bindings.Arguments.Primitive as Prim

getArgs : IO (List String)
getArgs = lift Prim.getArgs
```

I guess it's a good way to learn about the language's and the standard library's internals? Which, maybe, I did not want to. Anyway.

"Hand-crafted" solution

Now that we have access to the arguments, we just have to make sense of them. We use a type of options:

```
record grepOptions : Set where
  field
    -V     : Bool           -- version
    -v     : Bool           -- invert match
    -i     : Bool           -- ignore case
    regexp : Maybe String   -- regular expression
    files  : List FilePath  -- list of files to mine
open grepOptions public
```

And "hand-craft" a function populating it:

```
parseOptions : List String -> grepOptions
parseOptions args =
  record result { files = reverse (files result) }
  where
    cons : grepOptions -> String -> grepOptions
    cons opt "-v" = record opt { -v = true }
    cons opt "-V" = record opt { -V = true }
    cons opt "-i" = record opt { -i = true }
    cons opt str  =
      if is-nothing (regexp opt)
      then record opt { regexp = just str }
      else record opt { files  = str :: files opt }

    result : grepOptions
    result = foldl cons defaultGrepOptions args
```

A few issues

- I don't want to have to write this for every app
- I'm not even dealing with options yet
- This is not even ready for consumption yet!

```
regexp : Maybe String
```

What is a command-line interface?

- A set of *distinct* flag or options
- Each potentially coming with an *argument*
- Living in a *domain* of values
- We know how to *parse*

The (minimal) type of an `Argument`

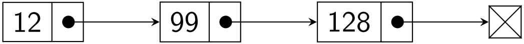
```
record Argument (l : Level) : Set (suc l) where
  field
    flag        : String
    domain      : Domain l
    parser      : parserType domain

data Domain (l : Level) : Set (suc l) where
  None :                          Domain l
  Some : (S : Set l)          -> Domain l
  ALot : (M : RawMagma l) -> Domain l

parserType : {l : Level} -> Domain l -> Set l
parserType None      = Lift Unit
parserType (Some S) = String -> String || S
parserType (ALot M) = String -> String || carrier M
```

A CLI is defined by an **extensible record** of arguments.

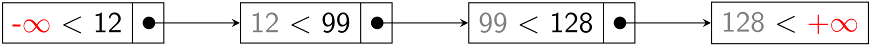
- guaranteed uniqueness of flags
- easy to lookup values
- easy to extend
- first class citizens (generic programming possible!)

The type of extensible records

McBride to the rescue: "How to keep your neighbours in order" tells us how to build in the invariant stating that a tree's leaves are sorted.
In the special case of linked lists, using a *strict* total order, we move from:



To the proven ordered:



Key ideas

Extend any ordered set with +/-infinity:

```
data [_] {l : Level} (A : Set l) : Set l where
  -infty :               [ A ]
  emb_   : (a : A) -> [ A ]
  +infty :               [ A ]
```

Define a type of ordered lists:

```
data USL (lb ub : Carrier) : Set _ where
  []     : lb < ub -> USL lb ub
  _,_::_ : hd (lt : lb < emb hd) (tl : USL (emb hd) ub) ->
           USL lb ub
```

Top level type: relax the bounds as much as possible!

```
Arguments = USL -infty +infty
```

As a consequence:

- really easy to write the correct:

  ```
  insertM : lb < x < ub -> USL lb ub -> Maybe (USL lb ub)
  ```

  rather than the intuitive:

  ```
  insertM : x -> USL lb ub -> Maybe (USL (min lb x) (max x ub))
  ```

- We can search for values satisfying a decidable property:

  ```
  search : (d : Decidable R) f (a : A) (xs : USL lb ub) ->
  Dec (el ** el inUSL xs ** R (f el) a))
  ```

where `_inUSL_` are fancy de Bruijn indices:

```
data _inUSL_ (a : _) : USL lb ub -> Set _ where
  z :                 a inUSL a , lt :: xs
  s : a inUSL xs -> a inUSL b , lt :: xs
```

The values of type an extensible record

Jon Sterling to the rescue: "Vinyl: Modern Records for Haskell" tells us what they should look like.

```
Mode : (args : USL lb ub) -> Set (suc l)
Mode args = arg (pr : arg inUSL args) -> Set l

options : (args : USL lb ub) (m : Mode args) -> Set l
options []                    m = Lift Unit
options (hd , lt :: args) m = m hd 0 * options args m'
```

Benefits

- Mode morphisms ⇒ record morphisms
- `get` through `search` + `lookup : (pr : x inUSL args) (opts : options args m) -> m x pr`
- generic parsing function!
  `parse : List String -> (args : Arguments) -> String || options args MaybeMode`
- generic usage function! `usage : Arguments -> String`

We can **run** an awful lot at **compile time**

- Type-rich structures internally
- Decidability on concrete instances externally (smart constructors)

For instance using `fromJust`:

```
fromJust : (a : Maybe A) {pr : maybe (\_ -> Unit) Void a}
           -> A
fromJust (just a) {pr} = a
fromJust nothing  {()}
```

we can turn `insertM : x (xs : Arguments) -> Maybe Arguments` into:

```
insert : x (xs : Arguments) {pr : _} -> Arguments
insert x xs = fromJust (insertM x xs)
```

Future Work

- Validation (DSL to write Mode morphisms?)
- Clean up the interface
- More parsers for base types
- Identify the utilities worth sending upstream