Radboud University

agdARGS

Declarative Hierarchical Command Line Interfaces

Guillaume Allais

Type Theory based Tools 2017 - Paris

Radboud University

Motivation: When it typechecks...

Ship it? More like shelve it! :(

Core algorithm

- Data structures with strong invariants
- Fully certified

\+ Boilerplate

- Validation of unsafe data
- (Command Line / Graphical) Interface

= Executable

Case study: a minimal `grep`

No access to the program's command-line arguments

- Add postulate + `COMPILED` pragma for `getArgs`
- Wrap in the `IO` monad

Ad-hoc parsing function

"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              -- invert match
    -i      : Bool              -- ignore case
    regexp  : Maybe String      -- regular expression
    files   : List FilePath     -- list of files to process
```

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 "-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
```

What is the specification of a CLI?

Types: What is a command-line interface?

What is a Command-Line Interface?

- A **description**
- A list of **subcommands**
- A list of **modifiers** (**flags** & **options**)
- Default **arguments**

What should we get from declaring one?

- The corresponding parser
- Usage information

Types: Example of a CLI

```
Grep = record
  { description = "Print lines matching a regexp"
  ; subcommands = noSubCommands
  ; arguments   = lotsOf filePath
  ; modifiers   =
    , "-v" ::= flag "Invert match"
    < "-i" ::= flag "Ignore case"
    < "-e" ::= option "Regexp" regexp
    < <>
  }
```

Internal representation

Extensible records

Represent field names as sorted lists

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

Associate a type to each field name

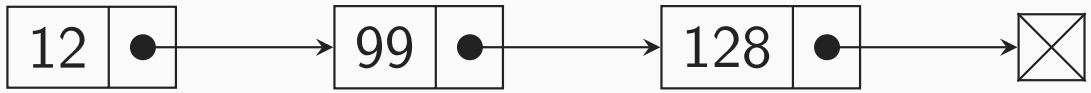
Generate record types by recursion on the list of field names

Remark: Drive type inference

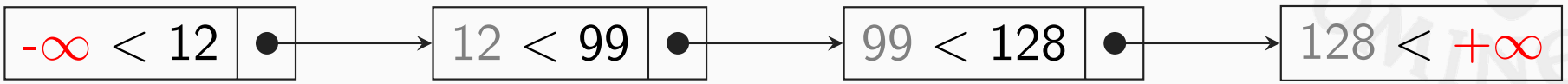
The type of extensible records

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 [_] (A : Set) : Set where
  -infty :              [ A ]
  emb_   : (a : A) -> [ A ]
  +infty :              [ A ]
```

Define a type of ordered lists:

```
data USL' (lb ub : [ A ]) : Set where
  []      : lb < ub -> USL' lb ub
  _,_::_ : hd -> lb < emb hd -> USL' (emb hd) ub ->
           USL' lb ub
```

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

```
type USL A = USL' (-infty : [ A ]) +infty
```

CLI

```
data Modifier name where
  mkFlag    : Record _ Flag   -> Modifier name
  mkOption  : Record _ Option -> Modifier name

record Command name : Set where
  inductive; constructor mkCommand
  field
    description : String
    subcommands : names ** Record names Command
    modifiers   : names ** Record names Modifier
    arguments   : Arguments
```

Design a nice interface

We can run an awful lot at compile time

Fully-explicit, invariant-heavy structures internally

vs. Decidability on concrete instances externally (smart constructors)

Remember:

```
, "-v" ::= flag "Invert match"
< "-i" ::= flag "Ignore case"
< "-e" ::= option "Regexp" regexp
< <>
```

We can run an awful lot at compile time

Fully-explicit, invariant-heavy structures internally

vs. Decidability on concrete instances externally (smart constructors)

Remember:

```
, "-v" ::= flag "Invert match"
< "-i" ::= flag "Ignore case"
< "-e" ::= option "Regexp" regexp
< <>
```

Using the smart constructors:

```
<>       : Record [] _
_::=_<_ : forall n -> S -> Record nms fields ->
            Record (insert n nms) (Finsert n nms S)
```

Generic Programming over Interfaces

Parsing

Parsing is decomposed in 3 phases

- subcommand selection
- modifier and arguments collection
- argument collection (triggered by "--")

And the returned result is *guaranteed* to respect the CLI:

```
parseCLI : (c : CLI) -> List String -> Error (ParsedCLI c)
withCLI  : (c : CLI) (k : ParsedCLI c -> IO a) -> IO a
```

Usage information

We know a lot about the structure of the interface. Let's use it!

```
usage : CLI -> String
```

e.g.

```
grep        Print lines matching a regexp
  -e          Regexp
  -i          Ignore case
  -v          Invert match
```

Conclusion

Conclusion

- Declarative
- Hierarchical
- Type-inference friendly
- Size-indexed internal representation
- Parser & Usage

Future Work

- Validation DSL (cf. Jon Sterling's Vinyl)
- Syntactic sugar for writing the continuation
  `(k : ParsedCLI c -> IO a)`
- Compound flags
- Other types of documentation (e.g. man pages)
- More parsers for base types
- Set level issues