Enforcing a discipline of Total Functional Programming through Dependent Types

Donovan Crichton

July 2024

Preliminaries

- Slides and Examples available at:
  https://github.com/donovancrichton/Talks
- This talk: BFPG/TotalFPThroughDepTypes

About me

- PhD Candidate
- Computing Foundations
- School of Computing

- Visiting Scholar
- Trusted Systems Lab
- IIIS

- ASD Co-Lab Scholar

- Veitch Lister Consulting
- Software Engineer

Alex (I): A Lexer Generator Library (GHC)

Character Sets and Macros

```
-- character sets
$digit  = 0-9
$small  = a-z
$big    = A-Z
$dash   = \-
$gt     = \>
$prime  = \'
$uscore = \_
$lambda = \\
$idchar = [$small $big $digit $prime $uscore]
-- character set macros
@bigid   = big idchar*
@smallid = small idchar*
@arrow   = $dash $gt
```

Alex (II): A Lexer Generator Library (GHC)

Regex Rules and Tokens

```
-- <state>     <regex>    {<func>}
   <0>         @arrow     {tokArrow}
   <0>         @bigid     {tokBigId}
   <0>         @smallid   {tokSmallId}
   <0>         $lambda    {tokLambda}
{
  data Token = TSmallId | TBigId
    | TArrow | TLambda

  tokLambda :: input -> Alex Token
  tokLambda input = pure TLambda

  tokArrow :: input -> Alex Token
  tokArrow input = pure TArrow
}
```

Alex (III): A Lexer Generator Library (GHC)

Output Token List

```haskell
  lex "\foo -> \bar -> \baz -> Baz foo bar"
-- gives us something like:
  [TLambda, TSmallId, TArrow,
   TLambda, TSmallId, TArrow,
   TLambda, TSmallId, TArrow,
   TBigId, TSmallId, TSmallId]
```

Happy (I): A Parser Generator Library (GHC)

Token Directive: Mirrors the Lexer

```
{
  import qualified Lexer as LEX
  import qualified ParseTree as AST
}
%token
  smallIdent {LEX.TSmallId}
  bigIdent   {LEX.TBigId}
  arrow      {LEX.TArrow}
  lambda     {LEX.TLambda}
```

Happy (II): A Parser Generator Library (GHC)

Production Rules and Sadness

```
Term :: {AST.Term}
  : smallIdent                  {parseVar $1}
  | bigIdent                    {parseDataCon $1}
  | lambda smallIdent
      arrow Term                {parseLambda $2 $4}
  | Term Term                   {parseApp $1 $2}
{
  parseVar :: LEX.Token -> AST.Term
  parseVar (TSmallId) = MkSmallRef
  parseVar _          = ?whatgoeshere

  parseDataCon :: LEX.Token -> AST.Term
  parseDataCon (TBigId) = MkDataCon
  parseDataCon _        = ?whatabouthere?
}
```

Awkward Pattern-Matches, A type is too large.

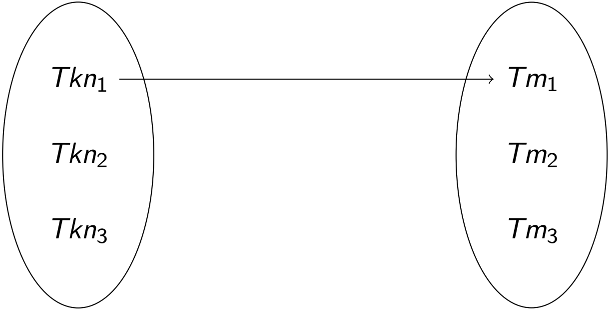
A quick note on the Algebra of Types.

Types

$Unit = \{*\}$
$Bool = \{True, False\}$
$Pair(A, B) = A \times B$
$Either(A, B) = A \sqcup B$
$Maybe(A) = \{*\} \sqcup A$
$A \to B = A \mapsto B$

Cardinality

$|Unit| = 1$
$|Bool| = 2$
$|Pair(A, B)| = |A| \times |B|$
$|Either(A, B)| = |A| + |B|$
$|Maybe(A)| = 1 + |A|$
$|A \to B| = |B|^{|A|}$

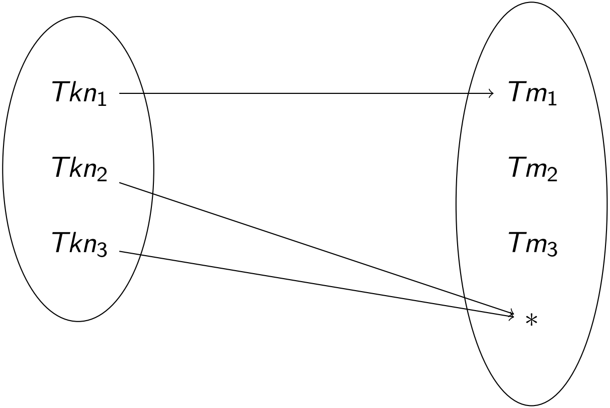


Overview of Totality

Totality

A function is total if it is *defined over all inputs*, and guaranteed to terminate.

```
f :: Token -> Maybe Term
```



Why do we want totality?

Compositionality

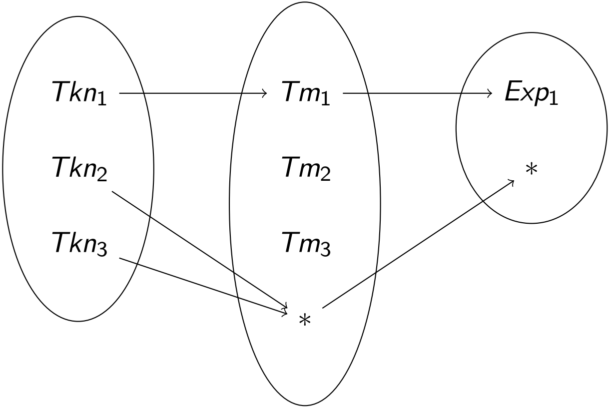
John Huges (1978) "Why Functional Programming Matters": Argues that compositionality is the backbone of modular programming.
Partial functions do not compose.

Larger codomains

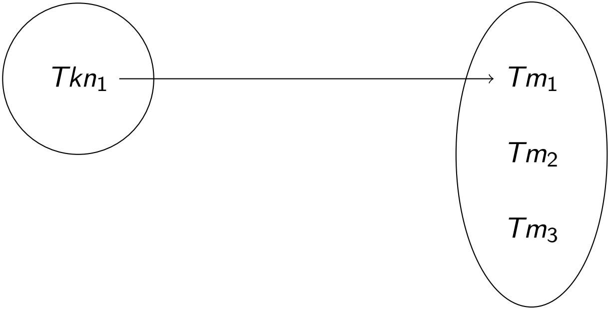
We can regain totality by increasing the size of our codomains, i.e with a Maybe type, but now we can only compose with functions that take Maybe types. This leads to a lot of unnecessary extending of domains.

An (unsatisfying) example of extending the codomain

```
f :: Token -> Maybe Term
g :: Maybe Term -> Maybe Exp
```



Ideally, we'd like to restrict the domain.



Fortunately, we can do this via dependent types.

Indexing our valid tokens by tokens.

Token and ValidToken Types

```
data Token =
  TBigId
  | TSmallId
  | TArrow
  | TLambda
  | TNull

data ValidToken : Token -> Type where
  VTNull     : ValidToken TNull
  VTBigId    : ValidToken TBigId
  VTSmallId  : ValidToken TSmallId
  VTArrow    : ValidToken TArrow
  VTLambda   : ValidToken TLambda
```

Indexing our valid tokens by tokens.

Token and ValidToken Types

```
-- null Token to satisfy totality checker.
total
match : String -> (a : Token ** ValidToken a)
match "foo" = (_ ** VTSmallId)
match "bar" = (_ ** VTSmallId)
match "baz" = (_ ** VTSmallId)
match "Foo" = (_ ** VTBigId)
match "=>"  = (_ ** VTArrow)
match "\\"  = (_ ** VTLambda)
match _     = (_ ** VTNull)
```

An introduction to dependent pairs.

Dependant Pairs (Σ types) in code.

```
data DPair : (a : Type) -> (a -> Type) -> Type
  MkDPair : {p : a -> Type}
          -> (fst : a) -> p fst -> DPair a p

fst : DPair a p -> a
fst (MkDPair x prf) = x

snd : {p : a -> Type}
   -> (rec : DPair a p) -> p (fst rec)
snd (MkDPair x prf) = prf
```

Using dependent pairs and GADTs to restrict our input domains.

Total Parsing with no maybes or errors

```
total
parseSmallId : ValidToken TSmallId -> Term
parseSmallId VTSmallId = MkRef

total
example : Term
example = parseSmallId (DPair.snd (match "foo"))
```