Functional Logic Design Patterns

Michael Hanus

Christian-Albrechts-Universität Kiel

joint work with

Sergio Antoy

Portland State University

SOME HISTORY AND MOTIVATION

1993 ([POPL'94, JACM'00]): **Needed Narrowing**

Good (optimal) evaluation strategy for functional logic programs

1995/96 ([ILPS'95, POPL'97]): **Design of Curry**

"Standard" functional logic language

needed narrowing $+$ residuation/concurrency

1999 ([FROCOS'00]): **Efficient implementation of Curry**

PAKCS: Portland Aachen Kiel Curry System

Since then: **various applications**

- ➜ What are the programming principles?
- ➜ What are interesting design principles?
- ➜ What are the advantages compared to purely functional or purely logic programming?

SOME HISTORY AND MOTIVATION

1993 ([POPL'94, JACM'00]): **Needed Narrowing**
Good (optimal) evaluation strategy for functional logic programs

1995/96 ([ILPS'95, POPL'97]): **Design of Curry**
"Standard" functional logic language
needed narrowing + residuation/concurrency

1999 ([FROCOS'00]): **Efficient implementation of Curry**
PAKCS: Portland Aachen Kiel Curry System

Since then: **various applications**

- ➜ What are the programming principles?
- ➜ What are interesting design principles?
- ➜ What are the advantages compared to purely functional or purely logic programming?

Some answers: this talk (ongoing work)

DESIGN PATTERNS

- good solution to recurring problems in software design
- not code but recipes to implement particular ideas
- reuse of ideas (not code)
- learn from experts
- introduced in object-oriented software development
- ideas also applicable to other paradigms

DESIGN PATTERNS

- good solution to recurring problems in software design
- not code but recipes to implement particular ideas
- reuse of ideas (not code)
- learn from experts
- introduced in object-oriented software development
- ideas also applicable to other paradigms

Functional logic design patterns:

learn to exploit integrated functional and logic programming features

FUNCTIONAL LOGIC PROGRAMMING

Approach to amalgamate ideas of declarative programming

- efficient execution principles of functional languages
  (determinism, laziness)
- flexibility of logic languages
  (constraints, built-in search)
- avoid non-declarative features of Prolog
  (arithmetic, I/O, cut)
- combine best of both worlds in a single model
  - ➜ higher-order functions
  - ➜ declarative I/O
  - ➜ concurrent constraints

CURRY

[Dagstuhl’96, POPL’97]

As a language for concrete examples, we use **Curry**:

- multi-paradigm language
  (higher-order concurrent functional logic language,
  features for high-level distributed programming)
- extension of Haskell (non-strict functional language)
- developed by an international initiative
- provide a standard for functional logic languages
  (research, teaching, application)
- several implementations available (e.g., PAKCS)

⇝ `http://www.informatik.uni-kiel.de/~curry`

VALUES

Values in imperative languages: basic types + pointer structures

Declarative languages: **algebraic data types** (Haskell-like syntax)

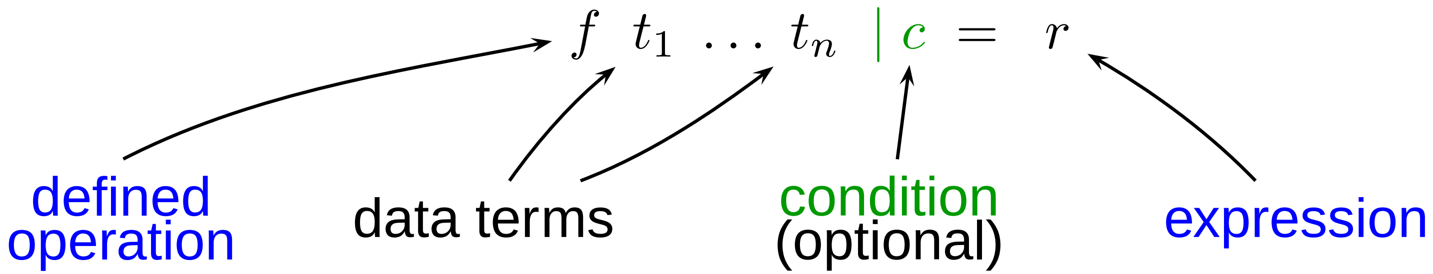
```
data Bool   = True   | False
data Nat    = Z      | S Nat
data List a = []     | a : List a      -- [a]
data Tree a = Leaf a | Node [Tree a]

data Int = 0 | 1 | -1 | 2 | -2 | ...
```

Value ≈ **data term**, **constructor term**:
well-formed expression containing variables and data type constructors

`(S Z)`   `1:(2:[])`   `[1,2]`   `Node [Leaf 3, Node [Leaf 4, Leaf 5]]`

CURRY PROGRAMS

Functions: operations on values defined by equations (or rules)



$$f\ \ t_1\ \ \ldots\ \ t_n\ \ |\ c\ \ =\ \ r$$

defined operation

data terms

condition (optional)

expression

```
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last xs | conc ys [x] =:= xs
        = x                       where x,ys free
```

`last [1,2]` $\rightsquigarrow$ `2`

EXPRESSIONS

$e ::=$

c	(constants)
x	(variables x)
$\texttt{(}e_0\ \ e_1 \ldots e_n\texttt{)}$	(application)
$\texttt{\textbackslash}x\ \texttt{->}\ e$	(abstraction)
$\texttt{if}\ \ b\ \ \texttt{then}\ \ e_1\ \ \texttt{else}\ \ e_2$	(conditional)

EXPRESSIONS

e ::=

c	(constants)
x	(variables x)
$(e_0\ \ e_1 \ldots e_n)$	(application)
$\backslash x\ \texttt{->}\ e$	(abstraction)
$\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	(conditional)
$e_1\texttt{=:=}e_2$	(equational constraint)
$e_1\ \ \texttt{\&}\ \ e_2$	(concurrent conjunction)
$\texttt{let}\ x_1,\ldots,x_n\ \texttt{free in}\ e$	(existential quantification)

EXPRESSIONS

e ::=

c	(constants)
x	(variables x)
$(e_0 \; e_1 \ldots e_n)$	(application)
$\backslash x$ `->` e	(abstraction)
`if` b `then` e_1 `else` e_2	(conditional)
e_1`=:=`e_2	(equational constraint)
e_1 `&` e_2	(concurrent conjunction)
`let` $x_1, \ldots, x_n$ `free in` e	(existential quantification)

Equational constraints over functional expressions:

`conc ys [x] =:= [1,2]` $\leadsto$ `{ys=[1],x=2}`

Further constraints: real arithmetic, finite domain, ports

EVALUATION

Naive approach: **Flattening**

- ➜ functional notation syntactic sugar for relations
- ➜ consider result value as additional (initially unbound) argument
- ➜ n-ary function $\leadsto$ $(n+1)$-ary predicate
- ➜ target language: Prolog

EVALUATION

Naive approach: **Flattening**

- ➜ functional notation syntactic sugar for relations
- ➜ consider result value as additional (initially unbound) argument
- ➜ n-ary function $\leadsto$ $(n+1)$-ary predicate
- ➜ target language: Prolog

```
conc([]    ,Ys,Ys).
conc([X|Xs],Ys,[X|Zs]) :- conc(Xs,Ys,Zs).

last(Xs,X) :- conc(Ys,[X],Xs).
```

EVALUATION

Naive approach: **Flattening**

- functional notation syntactic sugar for relations
- consider result value as additional (initially unbound) argument
- n-ary function $\leadsto$ $(n+1)$-ary predicate
- target language: Prolog

```
conc([]     ,Ys,Ys).
conc([X|Xs],Ys,[X|Zs]) :- conc(Xs,Ys,Zs).

last(Xs,X) :- conc(Ys,[X],Xs).
```

Disadvantage:

- some arguments not needed for computing the result
- functional dependencies not exploited by naive flattening
- wasting resources, not optimal

LAZY EVALUATION

- functions are lazily evaluated (evaluate only needed redexes)
- support infinite data structures, modularity
- optimal evaluation (also for *logic programming*)

Distinguish:

flexible (generator) and *rigid* (consumer) functions

Flexible functions $\leadsto$ logic programming

Rigid functions $\leadsto$ concurrent programming

FLEXIBLE VS. RIGID FUNCTIONS

```
f 0 = 2
f 1 = 3
```

rigid/flexible status not relevant for ground calls:

```
f 1   ⇝   3
```

`f` flexible:

```
f x =:= y   ⇝   {x=0,y=2} | {x=1,y=3}
```

`f` rigid:

```
f x =:= y   ⇝   suspend

f x =:= y & x=:=1   ⇝   {x=1} f 1 =:= y      (suspend f x)
                    ⇝   {x=1} 3 =:= y        (evaluate f 1)
                    ⇝   {x=1,y=3}
```

Default in Curry: constraints are flexible, all others are rigid

SET-VALUED FUNCTIONS

Rules must be constructor-based but not confluent:

- more than one rule applicable to a call
- set-valued (non-deterministic) functions
- more than one result on a given input

```
data List a = [] | a : List a

x ! y = x
x ! y = y

insert e []     = [e]
insert e (x:xs) = e : x : xs   !   x : insert e xs
```

SET-VALUED FUNCTIONS

Rules must be constructor-based but not confluent:

- more than one rule applicable to a call
- set-valued (non-deterministic) functions
- more than one result on a given input

```
data List a = [] | a : List a

x ! y = x
x ! y = y

insert e []     = [e]
insert e (x:xs) = e : x : xs   !   x : insert e xs

perm []     = []
perm (x:xs) = insert x (perm xs)
```

SET-VALUED FUNCTIONS

Rules must be constructor-based but not confluent:

- more than one rule applicable to a call
- set-valued (non-deterministic) functions
- more than one result on a given input

```
data List a = [] | a : List a

x ! y = x
x ! y = y

insert e []     = [e]
insert e (x:xs) = e : x : xs  !  x : insert e xs

perm []     = []
perm (x:xs) = insert x (perm xs)
```

perm [1,2,3] $\rightsquigarrow$ [1,2,3] | [1,3,2] | [2,1,3] | ...

FEATURES OF CURRY

Curry's basic operational model:

- ➜ conservative extension of lazy functional and (concurrent) logic programming
- ➜ generalization of concurrent constraint programming with lazy (optimal) strategy

FEATURES OF CURRY

Curry’s basic operational model:

- ➜ conservative extension of lazy functional and (concurrent) logic programming
- ➜ generalization of concurrent constraint programming with lazy (optimal) strategy

Further features for application programming:

- ➜ modules
- ➜ monadic I/O
- ➜ encapsulated search [PLILP’98]
- ➜ ports for distributed programming [PPDP’99]
- ➜ libraries for
  - GUI programming [PADL’00]
  - HTML programming [PADL’01]
  - XML programming
  - persistent terms
  - ...

Not relevant for our collection of design patterns

DESIGN PATTERNS VS. IDIOMS

No formal definition, but:

- ➜ idioms are more language specific
- ➜ idioms address smaller and less general problems

DESIGN PATTERNS VS. IDIOMS

No formal definition, but:

- ➜ idioms are more language specific
- ➜ idioms address smaller and less general problems

Example: copy a string in C:

```
while(*s++ = *t++) ;
```

Idiom solves simple problem and relies on specific properties of C

- ➜ strings end with null character
- ➜ *false* represented by integer `0`

DESIGN PATTERNS VS. IDIOMS

No formal definition, but:

- ➜ idioms are more language specific
- ➜ idioms address smaller and less general problems

Example: copy a string in C:

```
while(*s++ = *t++) ;
```

Idiom solves simple problem and relies on specific properties of C

- ➜ strings end with null character
- ➜ *false* represented by integer `0`

Design patterns are more general in applicability and scope

AN IDIOM IN CURRY

Ensure: a function returns a value only if value satisfies certain property

Define an auxiliary operator `suchthat`:

```
infix 0 ‘suchthat‘

suchthat :: a -> (a->Bool) -> a

x ‘suchthat‘ p  |  p x  =  x
```

AN IDIOM IN CURRY

Ensure: a function returns a value only if value satisfies certain property

Define an auxiliary operator `suchthat`:

```
infix 0 `suchthat`

suchthat :: a -> (a->Bool) -> a

x `suchthat` p  |  p x  =  x
```

Example application: n-queens puzzle

Check all permutations and return only the “safe” ones:

```
queens x = permute x `suchthat` safe
```

AN IDIOM IN CURRY

Ensure: a function returns a value only if value satisfies certain property

Define an auxiliary operator `suchthat`:

```
infix 0 `suchthat`

suchthat :: a -> (a->Bool) -> a

x `suchthat` p  |  p x  =  x
```

Example application: n-queens puzzle

Check all permutations and return only the “safe” ones:

```
queens x = permute x `suchthat` safe
```

⇝ “`suchthat`” idiom yields terser and more elegant code

Design patterns are more general

STRUCTURE OF DESIGN PATTERNS

Name: a basic name

Intent: the intention of this pattern

Applicability: where it can be used

Structure: the basic structure of the solution

Consequences: properties of applying this pattern

CONSTRAINED CONSTRUCTOR

Data constructors: create data

Defined operations: manipulate data

Constructors are passive: don't check for invalid data

CONSTRAINED CONSTRUCTOR

Data constructors: create data

Defined operations: manipulate data

Constructors are passive: don’t check for invalid data

Name	*Constrained Constructor*
Intent	prevent invoking a constructor that might create invalid data
Applicability	a type is too general for a problem
Structure	define a function that either invokes a constructor or fails
Consequences	invalid instances of a type are never created by the function

CONSTRAINED CONSTRUCTOR: EXAMPLE

Missionaries and Cannibals puzzle:

State: # missionaries, # cannibals, boat present? (on one side)

```
data State = State Int Int Bool
```

Initial: `State 3 3 True`

CONSTRAINED CONSTRUCTOR: EXAMPLE

Missionaries and Cannibals puzzle:

State: # missionaries, # cannibals, boat present? (on one side)

```
data State = State Int Int Bool
```

Initial: `State 3 3 True`

Function `move` checks for valid states before moving:

```
move (State m c True)
    | m>=2 && (m-2==0 || m-2>c) && (c==3 || m-2=<c)
    = State (m-2) c False                 -- move 2 missionaries
```

…and 9 other rules with similar complex guards…

CONSTRAINED CONSTRUCTOR: EXAMPLE (CONT'D)

Idea: constructor constrained to create only valid states

```
makeState m c b | valid && safe = State m c b
    where valid = 0<=m && m<=3 && 0<=c && c<=3
          safe  = m==3 || m==0 || m==c
```

CONSTRAINED CONSTRUCTOR: EXAMPLE (CONT'D)

Idea: constructor constrained to create only valid states

```
makeState m c b | valid && safe = State m c b
    where valid = 0<=m && m<=3 && 0<=c && c<=3
          safe  = m==3 || m==0 || m==c
```

Now, the definition of `move` becomes straightforward:

```
move (State m c True)
   = makeState (m-2) c False      -- move 2 missionaries
   ! makeState (m-1) c False      -- move 1 missionary
   ! makeState m (c-2) False      -- move 2 cannibals
   ! ...
```

Similarly: create only valid paths from initial state

CONSTRAINED CONSTRUCTOR

Name	*Constrained Constructor*
Intent	prevent invoking a constructor that might create invalid data
Applicability	a type is too general for a problem
Structure	define a function that either invokes a constructor or fails
Consequences	invalid instances of a type are never created by the function

Not available in functional languages:

if a function call fails, then the entire computation fails

SEARCH FOR SOLUTIONS

Search problem:

- search space
- look for elements satisfying particular properties
- search strategies

Avoid enumeration of all elements by defining solutions incrementally

SEARCH FOR SOLUTIONS

Search problem:

- ➜ search space
- ➜ look for elements satisfying particular properties
- ➜ search strategies

Avoid enumeration of all elements by defining solutions incrementally

Example: **Stagecoach**: finding path between cities

Topology of a problem: distance function between cities

```
distance Boston Chicago = 1500
distance Boston NewYork = 250
...
distance Denver LosAngeles = 1000
distance Denver SanFrancisco = 800
distance SanFrancisco LosAngeles = 300
```

STAGECOACH EXAMPLE

Task: find a path from Boston to Los Angeles

Solution: sequence of connected cities, first = Boston, last = Los Angeles

STAGECOACH EXAMPLE

Task: find a path from Boston to Los Angeles

Solution: sequence of connected cities, first = Boston, last = Los Angeles

Instead of enumerating all potential solutions: incremental construction

Partial solution: sequence of connected cities, first = Boston

Complete solution: partial solution with last = Los Angeles

Strategy: extend partial solution until complete solution reached

STAGECOACH EXAMPLE (CONT'D)

Extend a partial solution:

```
addCity (c:cs) | distance c c1 =:= d1
               = c1:c:cs                    where c1,d1 free
```

STAGECOACH EXAMPLE (CONT'D)

Extend a partial solution:

```
addCity (c:cs) | distance c c1 =:= d1
               = c1:c:cs                 where c1,d1 free
```

Specification of search problem has three components:

- `extend` a partial solution
- `initial` partial solution
- `complete` to check for completeness of solution

STAGECOACH EXAMPLE (CONT'D)

Extend a partial solution:

```
   addCity (c:cs) | distance c c1 =:= d1
                  = c1:c:cs                where c1,d1 free
```

Specification of search problem has three components:

- ➜ `extend` a partial solution
- ➜ `initial` partial solution
- ➜ `complete` to check for completeness of solution

Non-deterministic search function:

```
   searchNonDet :: (ps->ps) -> ps -> (ps->Bool) -> ps
   searchNonDet extend initial complete = solve initial
    where
       solve psol = if complete psol then psol
                                      else solve (extend psol)
```

Solve: `searchNonDet addCity [Boston] (\(c:_)->c==LosAngeles)`

STAGECOACH EXAMPLE (CONT'D)

Advantages:

- natural formulation of stepwise extension as set-valued function
- non-deterministic specifications are often simpler and more adaptable

STAGECOACH EXAMPLE (CONT'D)

Advantages:

- natural formulation of stepwise extension as set-valued function
- non-deterministic specifications are often simpler and more adaptable

  add eastbound connections:

```
addCity (c:cs) | distance c1 c =:= d1
               = c1:c:cs                where c1,d1 free
```

STAGECOACH EXAMPLE (CONT'D)

Advantages:

- natural formulation of stepwise extension as set-valued function
- non-deterministic specifications are often simpler and more adaptable

  add eastbound connections:

  ```
  addCity (c:cs) | distance c1 c =:= d1
                 = c1:c:cs                 where c1,d1 free
  ```

- apply other search strategies:

  ```
  searchDepthFirst addCity [Boston] (\(c:_)->c==LosAngeles)
  ```

INCREMENTAL SOLUTION

Name	*Incremental Solution*
Intent	compute solutions in an incremental manner
Applicability	a solution consists of a sequence of steps
Structure	non-deterministically extend a partial solution stepwise
Consequences	avoid explicit representation of the search space

EXAMPLE: REPRESENTATION OF GRAPHS

Basic datatypes in declarative programming: lists, trees

Often more natural: graph structures

Example: GUIs ≈ tree structure with dependencies

EXAMPLE: REPRESENTATION OF GRAPHS

Basic datatypes in declarative programming: lists, trees

Often more natural: graph structures

Example: GUIs ≈ tree structure with dependencies

Graphs as standard algebraic datatypes:

```
data Graph = Graph [Node] [Edge]

data Node = Node Int
data Edge = Edge Int Int
```

EXAMPLE: REPRESENTATION OF GRAPHS

Basic datatypes in declarative programming: lists, trees

Often more natural: graph structures

Example: GUIs ≈ tree structure with dependencies

Graphs as standard algebraic datatypes:

```
data Graph = Graph [Node] [Edge]

data Node = Node Int
data Edge = Edge Int Int
```

```
g1 = Graph [Node 1, Node 2, Node 3]
           [Edge 1 2, Edge 3 2, Edge 1 3, Edge 3 3]
```

EXAMPLE: REPRESENTATION OF GRAPHS

Basic datatypes in declarative programming: lists, trees

Often more natural: graph structures

Example: GUIs ≈ tree structure with dependencies

Graphs as standard algebraic datatypes:

```
data Graph = Graph [Node] [Edge]

data Node = Node Int
data Edge = Edge Int Int
```

```
g1 = Graph [Node 1, Node 2, Node 3]
           [Edge 1 2, Edge 3 2, Edge 1 3, Edge 3 3]
```

Composing graphs: `(addGraph g1 g1)` $\leadsto$ intended structure?

LOCALLY DEFINED GLOBAL IDENTIFIER

Solution: local definition of names → globally unique identifiers

Unbound local variables as identifiers:

```
g1 = Graph [Node n1, Node n2, Node n3]
           [Edge n1 n2, Edge n3 n2, Edge n1 n3, Edge n3 n3]
     where n1,n2,n3 free
```

Scope of `n1`,`n2`,`n3` local to `g1`

- ➜ `g1` is compositional (like lists, trees)
- ➜ `(addGraph g1 g1)` contains six different nodes

LOCALLY DEFINED GLOBAL IDENTIFIER

Solution: local definition of names → globally unique identifiers

Unbound local variables as identifiers:

```
g1 = Graph [Node n1, Node n2, Node n3]
           [Edge n1 n2, Edge n3 n2, Edge n1 n3, Edge n3 n3]
    where n1,n2,n3 free
```

Scope of `n1`,`n2`,`n3` local to `g1`

- `g1` is compositional (like lists, trees)
- `(addGraph g1 g1)` contains six different nodes

Instantiate node identifiers, e.g., for visualization tools:

```
finalizeGraph (Graph ns es) = Graph (numberNodes 1 ns) es
  where numberNodes _ [] = []
        numberNodes n (Node ni : nodes)
            | ni =:= n    -- assign unique identifier
            = Node ni : numberNodes (n+1) nodes
```

LOCALLY DEFINED GLOBAL IDENTIFIER

Name	*Locally Defined Global Identifier*
Intent	ensure that a local name is globally unique
Applicability	a global identifier is declared in a local scope
Structure	introduce local names as logic variables to be bound later
Consequences	local names are globally unique

Useful for GUI and HTML programming with compositional structures [PADL’00, PADL’01]

LOCALLY DEFINED GLOBAL IDENTIFIER

Name	*Locally Defined Global Identifier*
Intent	ensure that a local name is globally unique
Applicability	a global identifier is declared in a local scope
Structure	introduce local names as logic variables to be bound later
Consequences	local names are globally unique

Useful for GUI and HTML programming with compositional structures [PADL'00, PADL'01]

Not available in functional languages (lack of free variables)

⇝ imperative or non-compositional approaches to graph programming

IMPROVING GRAPH REPRESENTATIONS

Disadvantage of previous graph representation:

node identifiers are integers $\leadsto$ does not enforce unbound variables

IMPROVING GRAPH REPRESENTATIONS

Disadvantage of previous graph representation:

node identifiers are integers $\leadsto$ does not enforce unbound variables

Solution: hide type of identifiers with private constructor

```
module Graph(NodeId,...) where
...
data NodeId = NodeId Int    -- constructor not exported

data Node = Node NodeId
data Edge = Edge NodeId NodeId
```

Effect:

- ➜ definition of graph instances remain identical
- ➜ arguments of `Node` are always unbound variables

OPAQUE TYPE

Name	*Opaque Type*
Intent	ensure that values of a datatype are hidden
Applicability	define instances of a type whose values are unknown
Structure	wrap values with a private constructor
Consequences	values can only be denoted by free variables

Not available in functional languages (lack of free variables)

FINDING INJECTIVE INDEX-VALUE MAPPINGS

Example: Crypto-arithmetic puzzle

`SEND + MORE = MONEY` (Problem)

`9567 + 1085 = 10652` (Solution)

Task: finding *injective* mapping from indices (`S`, `E`,. . .) to values (digits)

FINDING INJECTIVE INDEX-VALUE MAPPINGS

Example: Crypto-arithmetic puzzle

`SEND + MORE = MONEY` (Problem)

`9567 + 1085 = 10652` (Solution)

Task: finding *injective* mapping from indices (`S`, `E`,...) to values (digits)

Our solution: instead of generate-and-test, compute it *concurrently*

FINDING INJECTIVE INDEX-VALUE MAPPINGS

Example: Crypto-arithmetic puzzle

`SEND + MORE = MONEY` (Problem)

`9567 + 1085 = 10652` (Solution)

Task: finding *injective* mapping from indices (`S`, `E`,. . .) to values (digits)

Our solution: instead of generate-and-test, compute it *concurrently*

Declare one variable for each letter: `vs,ve,vn,vd,vm,vo,vr,vy`

Set up constraints:

```
vd+ve    =:= c0*10+vy  &
vn+vr+c0 =:= c1*10+ve  &
ve+vo+c1 =:= c2*10+vn  &
vs+vm+c2 =:= c3*10+vo  &  c3 =:= vm
```

with carries: c_i `= 0!1`

`SEND + MORE = MONEY` (CONT'D)

Variables `vs,ve,...` initially unbound $\leadsto$ constraints suspend

Bind variables to digits so that mapping is injective

`SEND + MORE = MONEY` (CONT'D)

Variables `vs,ve,...` initially unbound $\leadsto$ constraints suspend

Bind variables to digits so that mapping is injective

Here: use an inverse mapping from values to variables identified by tokens

SEND + MORE = MONEY (CONT'D)

Variables `vs,ve,...` initially unbound $\leadsto$ constraints suspend

Bind variables to digits so that mapping is injective

Here: use an inverse mapping from values to variables identified by tokens

Inverse mapping $\approx$ store: initially: 10 free variables:

```
store = [s0,s1,s2,s3,s4,s5,s6,s7,s8,s9]
    where s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 free
```

Bind letters to digits (fails if not possible injectively):

```
digit token | store !! x =:= token = x
    where x = 0!1!2!3!4!5!6!7!8!9
```

```
vs = nzdigit 'S'
ve = digit   'E'
vn = digit   'N'
...
```

CONCURRENT DISTINCT CHOICES

Name	*Concurrent Distinct Choices*
Intent	ensure that a mapping from indexes to values is injective
Applicability	index-value pairs are computed concurrently
Structure	bind a unique token to a variable indexed by a value
Consequences	the index-value relation is an injective mapping

Not available in functional languages (lack of free variables)

Not available in pure logic languages
(lack of concurrency $+$ functional notation)

CONCLUSIONS

Functional logic design patterns

- a few patterns applicable in various situations
  - ➜ Constrained Constructor
  - ➜ Incremental Solution
  - ➜ Concurrent Distinct Choices
  - ➜ Locally Defined Global Identifier
  - ➜ Opaque Type
- intended for functional logic languages
- initial approach in this area
- will be extended. . .

More examples on functional logic patterns:

`http://www.cs.pdx.edu/~antoy/flp/patterns`

More infos on Curry:

`http://www.informatik.uni-kiel.de/~curry`