Demand-driven Search in Functional Logic Programs

Michael Hanus

RWTH Aachen

Pierre Réty

Université d'Orléans

Functional Logic Programming

Combine most important declarative paradigms:

- *Functional programming:*
  functions, equations, λ-calculus
  (lazy) deterministic reduction

- *Logic programming:*
  predicates, logical formulas, predicate logic
  constraint solving, search

Advantages:

- efficient deterministic reduction (if possible)
- flexibility of logic languages
- avoid non-declarative features of Prolog
  (arithmetic, I/O, cut)

Recent techniques:

- optimal evaluation strategies
- non-deterministic functions
- encapsulated search

$\Rightarrow$ **Modular implementation of demand-driven search**

$\Rightarrow$ **Application from music: chord composition**

Curry: A Truly Integrated Functional Logic Language

[Dagstuhl'96, POPL'97]

- multi-paradigm language, combines
  - functional programming
  - logic programming
  - concurrent programming
- based on an optimal evaluation strategy
- conservative extension of lazy functional and (concurrent) logic programming
- conditional (constrained) rules
- higher-order, non-deterministic functions
- polymorphic type system, modules
- declarative (monadic) I/O
- equational constraints
- encapsulated search, committed choice
- external functions and constraint solvers

Curry Programs

Values: *data terms* containing *constructors* and *variables* (≈ *Herbrand terms*): `(S x)` `[0,(S 0)]`

```
data Bool   = True | False
data Nat    = 0    | S Nat
data List a = []   | a : List a
```

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

$$f \;\; t_1 \;\ldots\; t_n \;\; | \; \{c\} \;\; = \;\; r$$

defined operation

data terms

constraint (conjunction of equations)

expression

```
    0 + y = y               0 ≤ y      = True
(S x) + y = S(x+y)      (S x) ≤ 0      = False
                        (S x) ≤ (S y)  = x≤y

append []     ys = ys
append (x:xs) ys = x : append xs ys

sub m n | {n + d = m}  = d  where d free
```

Evaluation: Computing Values

- reduce expressions to their values
- replace equals by equals
- apply reduction step to a subterm (redex)
  (rule's left-hand side must *match* the subterm)

```
    0 + y = y                 0 ≤ y       = True
(S x) + y = S(x+y)        (S x) ≤ 0       = False
                          (S x) ≤ (S y)   = x≤y
```

`(S 0)+(S 0)` $\rightarrow$ `S (0+(S 0))` $\rightarrow$ `S (S 0)`

Lazy strategy: select an outermost redex

`0+0 ≤ (S 0)+(S 0)`
$\rightarrow$ `0 ≤ (S 0)+(S 0)`
$\rightarrow$ `True`

$\leadsto$ evaluate only needed redexes

$\leadsto$ *functional programming*

From Functional Programming to Logic Programming

Functional programming: *values*, no free variables

Logic programming: *computed answers* for free variables

Operational extension:

instantiate free variables, if necessary

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

Evaluate `(f x)`: – bind `x` to `0` and reduce `(f 0)` to `2`, or:

– bind `x` to `1` and reduce `(f 1)` to `3`

Computation step: $\underbrace{\textbf{bind}}_{\textit{logic}}$ and $\underbrace{\textbf{reduce}}_{\textit{functional}}$

$$e \leadsto \underbrace{\{\sigma_1\}\ e_1 \mid \cdots \mid \{\sigma_n\}\ e_n}_{\text{disjunctive expression}}$$

Reduce:	`(f 0)`	$\leadsto$	`2`
Bind and reduce:	`(f x)`	$\leadsto$	`{x=0} 2 | {x=1} 3`

Compute necessary bindings with *needed* strategy

$\leadsto$ *needed narrowing* [Antoy/Echahed/Hanus POPL'94]

Properties of Needed Narrowing

[Antoy/Echahed/Hanus POPL'94]

- **Sound** and **complete** (w.r.t. strict equality)
- **Optimality:**
  1. **No unnecessary steps:**
     Each narrowing step is needed, i.e., it cannot be avoided if a solution should be computed.
  2. **Shortest derivations:**
     If common subterms are shared, needed narrowing derivations have minimal length.
  3. **Independence of solutions:**
     Two solutions σ and σ' computed by two distinct derivations are independent.
- **Determinism:**
  No non-deterministic step during the evaluation of ground expressions ($\approx$ functional programming)
- **Restriction: inductively sequential rules**
  (i.e., no overlapping left-hand sides)
- Extensible to
  - conditional rules [Hanus ICLP'95]
  - overlapping lhs [Antoy/Echahed/Hanus ICLP'97]
  - concurrent evaluation [Hanus POPL'97]

Non-deterministic Functions

Functions can have more than one result value:

```
choose x y = x
choose x y = y
```

`choose 1 2` $\leadsto$ `1 | 2`

Non-deterministic list insertion and *permutations*:

```
insert x []     = [x]
insert x (y:ys) = choose (x:y:ys)
                         (y:insert x ys)

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

`permute [1,2,3]` $\leadsto$

```
    [1,2,3] | [2,1,3] | [2,3,1] |
    [1,3,2] | [3,1,2] | [3,2,1]
```

Encapsulated Search

[Hanus/Steiner PLILP'98]

Technique to avoid global search (backtracking)
(non-backtrackable I/O, efficiency control,...)

Idea:
Compute until a non-deterministic step occurs,
then give programmer control over this situation
(generalization of Oz's search operator)

Search:

- solve constraint containing search variable
- bind search variable to different solutions
  $\Rightarrow$ abstract search variable: $\backslash x$->$\{c\}$
- primitive search operator:

```
try :: (a -> Constraint) -> [a -> Constraint]
```

```
try \x->{1=2}        ~> []                              failure
try \x->{[x]=[0]}    ~> [\x->{x=0}]                     success
try \x->{f x = 3}    ~> [\x->{x=0, f 0 = 3},
                          \x->{x=1, f 1 = 3}]
                                                     disjunction
```

Encapsulated Search: Search Strategies

`try \x->{c}`: eval., stop after non-deterministic step

Depth-first search: collect all solutions

```
all :: (a -> Constraint) -> [a -> Constraint]
all g = collect (try g)
where
 collect []            = []
 collect [g]           = [g]
 collect (g1:g2:gs) =
                   concat (map all (g1:g2:gs))
```

```
all \l->{append l [1] = [0,1]}
⇝  [\l->{l=[0]}]
```

Further search strategies:

- compute only first solution:

  ```
  once g = head (all g)
  ```

- `findall`, best solution search, parallel search, ...

Advantage: avoid combinatorial explosion by encapsulating independent subgoals

Programming Demand-driven Search

Prolog: generate-and-test:

```
psort(Xs,Ys) :- permute(Xs,Ys), ordered(Ys).
```

Functional programming: list comprehensions:

```
psort xs = [ys | ys<-perms xs, sorted ys]
```

Prolog with coroutining: test-and-generate

```
psort(Xs,Ys) :- ordered(Ys), permute(Xs,Ys).
```

(Problem: floundering, heuristics)

Functional logic programming: test-of-generate:

```
sorted []   = []
sorted [x]  = [x]
sorted (x:y:ys) | {x<=y = True}
                = x : sorted (y:ys)

psort xs = sorted (permute xs)
```

Advantages:

- demand-driven generation of solutions (due to laziness)
- same efficiency as coroutining
- no floundering
- modular program structure

Example: Demand-driven Search

```
sorted []    = []
sorted [x]   = [x]
sorted (x:y:ys) | {x<=y = True}
                = x : sorted (y:ys)

psort xs = sorted (permute xs)
```

`psort [5,4,3,2,1]`

$\leadsto$ `sorted (permute [5,4,3,2,1])`

$\leadsto^*$ $\underbrace{\texttt{sorted (5 : 4 : permute [3,2,1])}}_{\text{undefined: discard this alternative}}$ $\mid \cdots$

$\leadsto$ $\ldots$

Effect: Permutations of `[3,2,1]` are not enumerated!

A Musical Application: The Problem

Input: melody of some song

Output: appropriate accompaniment (sequence of chords) in harmony with the given melody

"in harmony": no big dissonances

For each bar: one chord (preferred) or two chords (if necessary)

Show *best choices* for each bar
(frequently: a major and a minor chord)

Melody: list of pairs (*note*,*duration*)

Output: list of alternatives for each bar (*duration*: 8)

```
F_maj/8  ||  A_min/4  D_min/4
```

A Musical Application: Our Solution

Harmony: depends on style of music
$\rightsquigarrow$ traditional songs

Local criterion for each chord:
sum of dissonances for each note not too large

Dissonance: if distance between a note and some chord note is 1/2 or 1 tone

Global criterion for each bar:
if one chord causes too large dissonance
$\Rightarrow$ split bar into two halves

Then: sum of dissonances should be better than single chord dissonance

$\Longrightarrow$
Search for best chords satisfying local and global criterion

A Musical Application: Our Implementation

Basic data types:

```
data Note = C | D | E | F | G | A | B | R
data Chord = C_maj | D_maj | ... | B_min
```

Non-deterministic generation of chords for each bar:

```
aChord = C_maj    aChord = F_maj    ...
one_chord_bar = [(aChord,8)]
two_chord_bar = [(aChord,4), (aChord,4)]
```

Compute all appropriate chords for a melody bar:

```
compute_bar mbar =
 if one_chord_solutions == []
 then bestOf (findall \x ->
        {checkBarDiss two_chord_bar mbar = x})
 else bestOf one_chord_solutions
where one_chord_solutions = findall \x ->
        {checkBarDiss one_chord_bar mbar = x}
```

check local and global criterion

Encapsulated search: avoid combinatorial explosion of solutions by combining all solutions for each bar (2 best solutions per bar $\Rightarrow 2^n$ solutions for n bars)

A Musical Application: “Sounds of Silence” (Simon/Garfunkel)

The melody (list of melody bars): `[`

```
[(R,2),(A,1),(A,1),(C,1),(C,1),(E,1),(E,1)], [(D,8)],
[(R,2),(G,1),(G,1),(B,1),(B,1),(D,1),(D,1)], [(C,8)],
[(R,2),(C,1),(C,1),(E,1),(E,1),(G,1),(G,1)],
[(A,2),(A,1),(G,5)],
[(R,2),(C,1),(C,1),(E,1),(E,1),(G,1),(G,1)],
[(A,2),(A,1),(G,5)],
[(R,2),(C,1),(C,1),(A,3),(A,1)],
[(A,2),(A,1),(B,1),(C,3),(C,1)],
[(C,1),(B,1),(A,2),(G,4)], [(R,2),(A,1),(G,1),(E,4)],
[(R,1),(A,1),(A,1),(C,1),(G,4)], [(R,1),(B,3),(C,1),(A,3)] ]
```

Result (Simon's chords are in red) $\approx$ 24576 solutions:

```
Amin/8
G/8 || Dmin/8
G/8 || G7/8
C/8 || Amin/8
C/8                       (Amin/4 C/4)
F/4 C/4 || F/4 Emin/4 || Amin/4 C/4 || Amin/4 Emin/4
C/8
F/4 C/4 || F/4 Emin/4 || Amin/4 C/4 || Amin/4 Emin/4
F/8 || Amin/8        (C/4 F/4)
F/8 || Amin/8
F/4 C/4 || F/4 Emin/4 || Amin/4 C/4 || Amin/4 Emin/4
C/8 || Emin/8 || Amin/8
C/8                       (Amin/4 C/4)
G/4 F/4 || G/4 Amin/4 || Emin/4 F/4 || Emin/4 Amin/4
```

Conclusions

Functional logic languages for search problems:

- non-deterministic functions ≈ *generators*
- demand-driven evaluation ≈ *demand-driven search*
- "test-of-generate" ≈ *modular implementation*

Encapsulation of search:

- declarative (monadic) I/O
- avoid combinatorial explosion by combining independent subproblems

Application: composition of accompaniments

More infos on Curry:

`http://www-i2.informatik.rwth-aachen.de/~hanus/curry`