Functional Logic Programming

Kristjan Vedel

Imperative vs Declarative
Algorithm = Logic + <u>*Control*</u>

Imperative

- How?
- *Explicit* Control
- Sequences of commands for the computer to execute

Declarative

- What?
- *Implicit* Control
- Steps to execute specified by language implementation

Declarative Programming
Subparadigms

- Functional programming
  - Building blocks: (higher order) functions
  - Computation model: unidirectional, deterministic reduction (λ-calculus, combinatory logic)
- Logic (relational) programming
  - Building blocks: predicates/relations, logical variables
  - Computation model: multidirectional, non-deterministic search (resolution for horn clauses, 1st order logic)
- Constraint programming
- Other examples: SQL, DSLs (Make), Regexps etc

Functional Programming

- Program - collection of function definitions
- Computation by term rewriting.
- First-class and higher-order functions
- Recursion.
- Avoid mutable data
- Prominent (research) language: Haskell
- Deterministic and unidirectional (in contrast to logic prog.)

Functional Programming

- Common reduction strategies: leftmost-outermost with call-by-value or call-by-need
- λ-calculus reduction example (rightmost-first reduction)

```
(λx.x+1)((λy.y+2)3) =>> 6
```

```
apply         =>   apply     =>  apply   =>  +  =>  6
    /     \             /   \         /   \      / \
   λ       apply       λ     +       λ     5    5   1
  / \       /  \      / \   / \     / \
 x   +     λ    3    x   + 3   2   x   +
    / \   / \           / \           / \
   x   1 y   +         x   1         x   1
            / \
           y   2
```

Logic Programming
Resolution and Unification

- Prominent language: Prolog
- Commonly based on resolution of **Horn clauses** (at most one positive literal), subset of 1st order logic
- **Resolution** - proof by refutation, negation of goal in conjunction with knowledge base leads to contradiction
- **Unification** - unify(f(x,B), f(A,y)) --> f(A,B)
- **Search** (backtracking, *nondeterminism*)

Logic Programming
Unification example

likes(John, Jane).
likes(y, Jim).
likes(y, friend(y)).

- Substitution

UNIFY(likes(John,x), likes(John,Jane)) = {x/Jane}
UNIFY(likes(John,x), likes(y,Jim)) = {x/Jim, y/John}

- Substitution makes following two sentences identical:

UNIFY(likes(John,x), likes(y,friend(y))) = {x/John, x/friend(John)}

- Can't substitute ground term with another

UNIFY(likes(John,x), likes(x,Elizabeth)) = fail

- Variable may not occur in the term it is being unified with.

UNIFY(x, F(x)) = fail

Logic Programming
Resolution example

{C,**¬A**,¬B},{**A**},{B},{¬C}→{C,**¬B**},{**B**},{¬C}→**{C}**,{**¬C**}→**{}**

Prolog-style:

C :- A, B.

A.

B.

Goal: C

- Proof by refutation: add "not C" and resolve to empty set i.e. show contradiction
- Use unification where necessary

Constraint Programming

"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the *user states the problem, the computer solves it*." - E.Freuder

Constraint Programming

- Model and solve a problem by specifying constraints that fully characterize the problem. Featuring:
    - **Variables** that range over **domains**
    - Relations between variables stated as **constraints**.
    - **Constraint satisfaction**
        - (finite domains, combinatorial techniques)
    - **Constraint solving**
        - (infinite/complex domains, mathematical techniques)
- Constraint Satisfaction Problem (CSP)
    - Examples: Sudoku, map coloring, etc
- Usually in form of constraint logic programming
  also: functional+constraints, imperative+constraints

Constraint Programming

- Constraints over specific domain:
  - *boolean*, true/false constraints (SAT problem)
  - *integer*, *rational*
  - *linear*, for linear functions only
  - *finite*, constraints are defined over finite sets
  - *mixed*
- Solvers: systematic search
  - Generate and test
  - Backtracking
- Improvements to systematic search, heuristics, simplex, various other domain-specific solvers etc

Logic Programming (LP)
and Constraint Logic Programming (CLP)

LP languages (like Prolog) can be viewed as a subset of CLP with:

- ground terms (variables, constants, function symbols)
- single constraint "=" (syntactic equality)
- resolution algorithm for solving
  - backtracking
  - generate and test

Constraint Programming

Example

$$\begin{array}{rccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

```
% Prolog + CLPFD constraint solver library.
:- use_module(library(clpfd)).
sendmore(Digits) :-
   Digits = [S,E,N,D,M,O,R,Y],  % Create variables
   Digits ins 0..9,              % Associate domains to variables
   S #\= 0,                      % Constraint: S must be different from 0
   M #\= 0,
   all_different(Digits),        % all elements must take different values
            1000*S + 100*E + 10*N + D     % Other constraints
            + 1000*M + 100*O + 10*R + E
   #= 10000*M + 1000*O + 100*N + 10*E + Y,
   label(Digits).                % Start the search
```

Hybrid
Functional + Logic + Constraint

- Functional:
  - Efficiency
    - (Deterministic) reduction of function expressions
    - More *control* compared to logic programming
- Logic:
  - Expressive power:
    - Logical variables
    - Built-in search
- Constraint:
  - Expressiveness, efficient solving strategies
- **Narrowing** and/or **Residuation** techniques

Logic + Functional

- Extend logic language with functional concepts
- Example: Mercury
  - Based on Prolog
  - Strong, static, polymorphic types
  - Explicit determinism system
  - Closures, Currying, and Lambda expressions.

Functional + Logic

- Extend functional language with logic concepts
- Example: **Curry**

Curry
Language overview

- General purpose functional logic programming language.
- **Functional**: (deterministic) reduction of nested expressions, higher-order functions, lazy evaluation
- **Logic**: logical variables, partial data structures, built-in search
- **Constraint:** constraint structure, solvers
- **Concurrent**: Concurrent evaluation of constraints with synchronization on logical variables
- Syntax is mostly Haskell
  - Missing type classes :((experimental support exists)
  - Added mainly "*where x free*" for logical variables

Curry
Search for solutions

- **Logic variable** - variable in the condition and/or right-hand side of a rewrite rule which does not occur in the left-hand side:
  - **x** == 2 + 2 **where x free**
  - path a z = edge a **b** && path **b** z **where b free**
- **Search for solutions** - compute values for the arguments of functions so that the functions can be evaluated
  - Instantiates logic variables
  - Compute all possible solutions, one at a time

Curry

Search for solutions : Example

1. Prelude> x &&(y || (not x)) **where x,y free**
   Free variables in goal: x, y
   Result: True
   Bindings:
   x=True
   y=True ? ;
2. Result: False
   Bindings: x=True
   y=False ? ;
3. Result: False
   Bindings: x=False
   y=y ? ;
4. No more solutions.

Curry
Non-deterministic functions

- Non-deterministic insert

  insert :: a -> [a] → [a]

  insert x [] = [x]

  insert x (y:ys) = x : y : ys

  insert x (y:ys) = y : insert x ys
- Multiple result values:

  coin :: Int

  coin = 0

  coin = 1
- Calls to non-deterministic functions?

  coin + coin
- (?) :: a -> a -> a -- choice operator from Prelude

Curry
Constraints

- Types:
  - success :: Success
    - no visible literal values
    - denotes result of successfully solved constraints
- Operators:
  - Constrained equality
    - (**=:=**) :: a -> a -> Success
  - Parallel conjunction
    - (**&**) :: Success -> Success -> Success
  - Constrained expression
    - (**&>**) :: Success -> a -> a

Curry
Operational semantics

- Lazy evaluation of expressions with possible instantiation of free variables in expression
  - ground expressions -> as lazy functional language
  - instantiations -> as in logic programming
- **Answer expression**:
  - substitution σ + expression: e (Example: {x=0,y=2}2)
  - solved if e is data term
- **Disjunctive expression**:
  - Multiset of answer expressions
  - $\{ \sigma_1 e_1 \mid \sigma_2 e_2 \mid \ldots \mid \sigma_n e_n \}$
- **Computation step**: reduction in exactly one unsolved answer expression: $\{ \underline{\boldsymbol{\sigma_1 e_1}} \mid \sigma_2 e_2 \mid \ldots \mid \sigma_n e_n \}$

Curry
Operational semantics

- Examples:
  ```
  f 0 = 2
  f 1 = 3
  ```
  - f 1 evaluates to 3
  - f x evaluates to disjunctive expression { {x=0}2 | {x=1}3 }.
- Value is *demanded*
  - in argument of a function call if the left-hand side of some rule has a constructor at this position.
  - case expressions
  - arguments of external functions
- Free variables can occur where value is demanded. Solutions:
  - **Residuation**
  - **Narrowing**

Curry
Residuation

- **Residuation** - suspend function calls until they are ready for deterministic evaluation (free logic variable is bound)
    - incomplete - unable to compute solutions if arguments of functions are not sufficiently instantiated during computation
- Example:
    - Primitive arithmetic operators
    - Boolean equality: **(==) :: a -> a -> Boolean**
    - Prelude> x **==** 2+2 where x free

      Free variables in goal: x

      *** Goal suspended!

Curry
Narrowing

- **Narrowing** - combination of unification for parameter passing and reduction as evaluation mechanism
  - variable is bound to a value selected from among alternatives imposed by constraints.
  - complete in functional sense (normal forms computed if exist)
  - complete in logic sense (solutions computed if exist)
- <u>Example</u>:
  - Equational constraint: **(=:=) :: a -> a -> Success**
  - Prelude> x **=:=** 2+2 where x free

    Free variables in goal: x

    Result: success

    Bindings: x=4 ?

Curry
Rigid and flexible operators

- **Rigid** - operators that residue
    - most primitive operators (arithmetic etc)
- **Flexible** - operators that narrow
    - all defined operators
- For ground expressions (without logic variables) - no difference whether flexible/rigid
- *ensureNotFree* - primitive op to evaluate argument and suspend if logic variable

Curry
Example

- Prelude> x **++** [3,4] =:= [1,2,3,4] where x free
  Free variables in goal: x
  Result: success
  Bindings:
  x=[1,2]
  More solutions? [Y(es)/n(o)/a(ll)] y
  No more solutions.
- Prelude> x **+** 2 =:= 4 where x free
  Free variables in goal: x
  *** Goal suspended!

Curry
Evaluation example

2+x =:= y & f x =:= y

-->

{x=1} 2+1 =:= y & 3 =:= y

-->

{x=1,y=3} 2+1 =:= 3

-->

{x=1,y=3} 3 =:= 3

-->

{x=1,y=3}

- another solution: {x=0, y=2}

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

Curry
More examples

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

Used materials

- Antoy, Sergio. "Curry A Tutorial Introduction." (2007). http://www-ps.informatik.uni-kiel.de/currywiki/documentation/tutorial
- Antoy, Sergio, and Michael Hanus. "Functional logic programming." Communications of the ACM 53.4 (2010): 74-85.
- Apt, Krzysztof. Principles of constraint programming. Cambridge University Press, 2003.
- Hanus, Michael, et al. "Curry: An integrated functional logic language (version 0.8.3)." (2012).
- Hedman, Shawn. A first course in logic. Oxford: Oxford University Press, 2004.
- Lloyd, John W. "Practical advantages of declarative programming." Joint Conference on Declarative Programming, GULP-PRODE. Vol. 94. 1994.
- wikipedia.org :)