TinyProlog: Logic programming language

F# tricks and step-by-step guide

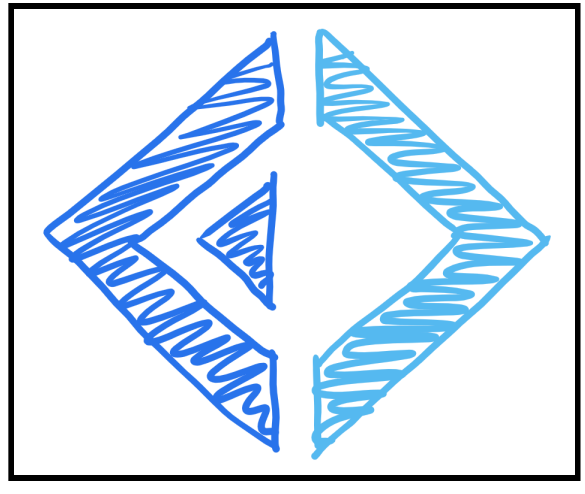
Tomas Petricek, Charles University

tomas@tomasp.net
@tomasp.net
https://tomasp.net
https://d3s.mff.cuni.cz/teaching/nprg077

Advanced F# features

Active patterns

- Custom patterns for use in `match`
- Match number with `Odd` or `Even`
- Recognize special forms of terms
- Complete or partial patterns



Sequence expressions

- Write code that generates a sequence of items
- Comprehensions (Haskell), generators (JS), ...
- Lazy `seq {..}` or eager `[..]` or arrays `[|..|]`

Charles University

Demo

Advanced F# features

TinyProlog

Code structure

```
(* Recursive term definition *)
type Term =
  | Atom of string
  | Variable of string
  | Predicate of
      string * Term list
  | Call of Term * Term list

(* Facts have empty Body *)
type Clause =
  { Head : Term
    Body : Term list }

(* Create a fact clause *)
let fact p =
  { Head = p; Body = [] }

(* Create a rule clause *)
let rule p b =
  { Head = p; Body = b }
```

TinyProlog programs

Encoded as F# types!

Atom vs. variable

Atom is a single data item, thing that exists.

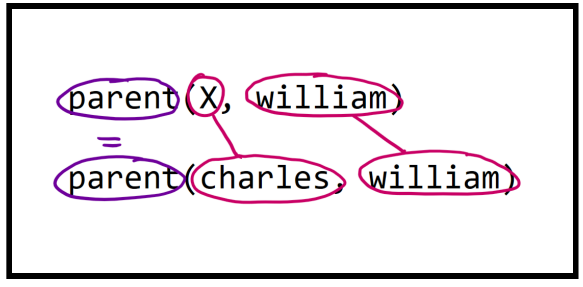
Variable is a place-holder that we want to assign a term to.

Charles University

The unification process

Tiny implementation

- Similar to our type inference code!
- **`unify`** and **`unifyLists`** functions
- Generate substitution for variables



Used in Prolog context

- Same 2 uses of substitution
- Occurs check done optionally
- Use fresh set of variables when reusing rules from program database!

Charles University

Unification logic

```
let rec unifyLists l1 l2 =
  match l1, l2 with
  | [], [] ->
      (* empty substitution*)
  | h1::t1, h2::t2 ->
      match unify h1 h2 with
      | Some(s) -> (*
         1. substitution 's' to
            unify 'h1' and 'h2'
         2. now unifiy 't1' and 't2'
            recursively & compose
         3. if not possible, fail *)
      | _ -> (* fail *)
  | _ -> (* fail *)

and unify t1 t2 =
  match t1, t2 with
  | Atom(a1), Atom(a2) -> (* does 'a1' match 'a2'? *)
  | Variable(v), t | t, Variable(v) ->
      (* return a substitution *)
  | Predicate(p1, args1), Predicate(p2, args2) ->
      (* if p1 = p2, unify arguments recursively *)
  | _ -> None
```

Split into two functions for better readability

unify matches terms

unifyLists matches two lists using **unify**

Charles University

```
% Number: 0
zero

% Number: 1
one = s(zero)

% Number: 5
five = s(s(s(s(s(zero)))))

% Empty list
empty

% List [1]
cons(one, empty)

% List [1; 5]
cons(one, cons(five, empty))
```

Adding support for numbers and lists

Nothing extra is needed!

Good enough for a tiny implementation.

Terribly inefficient and limited if you want to calculate anything!

Lab overview

TinyProlog system step-by-step

TinyProlog - Basic tasks

1. Implementing basic unification of terms
   Recursively match atoms, variables and predicates
2. Composing and applying substitutions
   To handle multiple occurrences of a variable correctly
3. Searching clauses & variable renaming
   Find applicable rules and relevant facts in program
4. Generating and proving goals recursively
   The key trick! Generate and solve goals in a loop
5. Adding numbers to TinyProlog
   Representing, calculating and pretty printing

TinyProlog - Bonus and super tasks

1. Lazy search and support for lists
   Refactoring for readability and more pretty printing
2. Generating magic squares in TinyProlog
   In which we find out how slow our implementation is :-)
3. Implementing call and functional maplist
   Adding special predicate for higher-order programming
4. Adding support for occurs checks
   If you want to make it slower and more correct
5. Implementing Prolog-style cut operator
   Super-bonus if you are into Prolog programming...

Lessons learned

A tiny logic programming language

- Remarkably similar to ML type inference!
- This is not a coincidence…
- Evaluation as search, not a sequence of steps
- Much work needed to make this practical

Charles University