Logic programming and types: An introduction to Mercury

Sergei Winitzki

SF Types, Theorems, and Programming Languages meetup

April 18, 2016

Logic programming as a DSL for logic puzzles (Prolog)

All jumping creatures are green. All small jumping creatures are martians. All green martians are intelligent.
Ngtrks is small and green. Pgvdrk is a jumping martian.
Who is intelligent? (inpired by S. Lem, *Invasion from Aldebaran*)

```
small(ngtrks). green(ngtrks).
martian(pgvdrk). jumping(pgvdrk).
green(X) :- jumping(X).
martian(X) :- small(X), jumping(X).
intelligent(X) :- green(X), martian(X).
?- intelligent(X).
X = pgvdrk
```

- The runtime will perform a backtracking search for a proof
- Classical logic limited to Horn clauses = executable program

Mercury in a nutshell: a Prolog dialect with static types

- Mercury borrows these from Prolog: logical variables, predicates, rules
- Syntax and semantics is very close to Prolog
- Data structures: list, queue, hash map
- Flexible mode system, type checking, mode and type inference
- Polymorphic types and type classes, type quantifiers
- Termination analysis for predicates
- Functions as a primitive, on par with predicates
- Higher-order functions and higher-order predicates
- Compilation to C, Java, Erlang, .NET
    - Interop and type-level compatibility with these environments

Modes, determinism, types, I/O

- Predicate arguments must have **types**, **modes**, and **determinism**
- "Martians" requires a declaration of modes & determinism for `main`:

```
pred main(io::di, io::uo) is cc_multi.
```

- Input/output: Special syntax `!IO` means a pair of `io` ("world") values
- Proof search may backgrack - the goal `intelligent(X)` may fail, so:

```
main(!IO) :- ( if intelligent(X) then write(X, !IO)
  else write_string("No solution", !IO) ).
```

- Determinism: `det`, `semidet`, `multi`, `nondet`, `failure`, `cc_multi`, `cc_nondet`
- Predefined modes: `in`, `out`, `di`, `uo`
  - A **mode** describes what happens to an LV during proof search:

    ```
    :- mode out == (free >> ground).
    :- mode uo == (free >> unique).
    :- mode di == (unique >> dead).
    ```

  - User-defined modes are possible

Example: integer factorial

- Implementation as a predicate:

```
:- pred fact(int::in, int::out) is det.
fact(N,F) :- ( if N =< 1 then F = 1 else fact(N-1, A), F = A*N ).
```

- Implementation as a function:

```
:- func fct(int) = int.
fct(N) = ( if N =< 1 then 1 else N*fct(N-1) ).
```

- The generated code is *identical* in both cases!
  - However, predicates are **not** "functions returning Boolean"
- Type/mode/deteterminism are inferred and statically checked
- *Order of goals* is inferred from modes rather than from code!

```
:- pred fact(int::in, int::out) is det.
fact(N,F) :- ( if N =< 1 then F = 1 else F = A*N, fact(N-1, A) ).
```

Static determinism

- Compilation fails if we define the predicate by clauses:

```
:- pred fact(int::in, int::out) is det.
fact(1,1).
fact(N,F) :- fact(N-1, A), F = A*N.
```

- Error message:

```
In ‘fact’(in, out): error: determinism declaration not satisfied.
Declared ‘det’, inferred ‘multi’.
Disjunction has multiple clauses with solutions.
```

- Similar error when defining a function by clauses

```
Error: invalid determinism for ‘fct’(in) = out:
the primary mode of a function cannot be ‘multi’.
```

- This is why we need to use a less readable `if-then-else`

Functions and expressions

- Functions are deterministic predicates with “in-out” mode
- Expressions are syntactic sugar for goals involving functions
- Hence, the general syntax for functions:
  `fname(Arg1, Arg2, ...)  = Result :- goal, goal, ..., Result = ...`
- “Lambda” predicate terms and function terms:

```
F1 = ( pred(N::in, X::out) is det :- fact(N, X) ).
F2 = fact . % same value as F1 by η-equivalence
F3 = ( func(N) = X :-  X = fct(N) ).
F4 = fct.   % same value as F3 by η-equivalence
```

- Higher-order predicates:
  `map(fact, [1,3,5], Res)` will unify `Res` with `[1,6,120]`
  - Declaring higher-order predicate modes is verbose and complicated

Algebraic types and polymorphism

- Primitive types: `char`, `int`, `float`, `string`
  - No "symbol literals" as in Prolog
- Predicate types, function types
  - Type syntax: `pred(int::in, int::out)` and `func(int) = int`
- Tuples, unions

```
:- type mytype3 ---> point({int,int},string); ok(string); failed.
:- type list(T) ---> []; [ T | list(T) ]. % special syntax
:- type tree(T) ---> leaf; branch(tree(T), T, tree(T)).
```

- Records with accessors

```
:- type employee ---> employee(name :: string, id :: int).
X = employee(...), if X ^ name = "myself" then ...
```

Example: "reversible computation"

- Convert integers between unary encoding and native representation

```
:- type unary ---> z; s(unary). %   z;  s(z);  s(s(z));  etc.
:- pred unary_int(unary, int).
:- mode unary_int(in, out). % is det.
:- mode unary_int(out, in). % is multi.
unary_int(C, N) :- (
  C = z, N = 0 ;
  C = s(B), N = M+1, unary_int(B, M)
% goals will be reordered here depending on mode!
).
```

- The same code can unify unary_int(s(s(z)), N) or unary_int(C, 3)
  - Determinism and mode of predicates is inferred from use!
  - *Different code* is compiled for each declared mode of unary_int(C, N)

Type classes à la Haskell

- ***

Existential types

- ***

Limitations of Mercury

- No higher-order type constructors (e.g. cannot define “traversable functor” parameterized by “applicative functor”)
- No row polymorphism for records
- No partially instantiated values (e.g. difference lists)

Summary

- How to unite "logic programming" and "functional programming"
- Mercury = Prolog + modes + types
- Features of the Mercury programming language (online link)