Funktional-logische Programmierung mit Mercury

Henning Thielemann

2008-02-11

Ziele

Verbesserung logischer Programmierung in Richtung

- mehr Sicherheit
- mehr Effizienz

im Handel gegen

- Einfachheit

Mercury = Mehr Curry?

Paradigmen

- Logisch: Klauseln,
  auch Modul-, Typsystem und Übergabemodi werden durch Klauseln gesteuert
  (Warum eigentlich nicht mit Fakten sondern mit Beweiszielen?)
- Funktional: Funktionen sind Werte,
  nicht Terme wie in PROLOG
  (Funktionen können rekursive Aufrufe enthalten, Terme nicht)

Seiteneffektfreiheit, Zustandsfreiheit

- Unikat-Modi für saubere Ein- und Ausgabe
- Dynamische Datenstrukturen in Bibliothek statt `assert/retract`

Parameterübergabemodi

- Pflichtangabe:
  erlaubte Muster für freie und gebundene Parameter
- Ausdrücke entweder vollständig frei oder vollständig gebunden, genau wie in Turbo-PROLOG
  d.h. `X` kann durch `predicate` an `(1,2)` gebunden werden, oder frei bleiben (`X`), aber nicht teilweise gebunden werden, etwa `(1,Y)`.

Typsystem

ähnlich zu Haskell 98

- Statisch typisiert für Fehlererkennung zur Übersetzungszeit
- polymorph (Typvariablen)
- automatische Ableitung von Typen (type inference)
- Typklassen
- ...auch mit mehreren Typparametern
- ...auch mit funktionalen Abhängigkeiten
- Funktionen höherer Ordnung
- Typeinschränkungen für Löser mit Nebenbedingungen (constraint solver) (experimentell)

Unterstützung für Funktionen

- Mercury unterstützt Funktionen als solche.
- Alternativen in PROLOG: deterministische Prädikate, Terme
- PROLOG: nur Prädikate können rekursiv sein

Fibonaccizahlenberechnung als Prädikat

```
:- pred fib(int::in, int::out) is det.

fib(N, X) :-
 ( if N =< 2
     then X = 1
     else fib(N - 1, A), fib(N - 2, B), X = A + B
  ).
```

if mit Prädikaten: `(if B then predicateX else predicateY)`

Fibonaccizahlenberechnung als Funktion

```
:- func fib(int) = int.

fib(N) = X :-
 ( if N =< 2
     then X = 1
     else X = fib(N - 1) + fib(N - 2)
  ).

fib(N) = ( if N =< 2 then 1 else fib(N-1) + fib(N-2) ).
```

if mit Werten: `(if B then valueX else valueY)`

Funktionen: Beobachtung

- Funktionen reduzieren die Variablen für Zwischenergebnisse
- `if-then-else` sowohl als Aussage als auch als Ausdruck, immer geklammert, `else` erforderlich

Verschiedene denkbare Techniken

- Ein- und Ausgabeströme (streams)
- Fortsetzung (continuation)
- Monade: Haskell 98
- Unikat-Modi (uniqueness types): Mercury, lineare Typen: Clean

Unikat-Modi

- Jeder erzeugte Unikat-Wert darf nur einmal weiterverwendet werden
- Optimierung:
  Verändern von Werten in der gleichen Speicherzelle
  (in-place update)
- sichere Ein- und Ausgabe

Verstecken von Variablen für Zwischenergebnisse

ohne syntaktischen Zucker

```
main(IOState_in, IOState_out) :-
 io.write_string("Hello, ", IOState_in, IOState_tmp1),
 io.write_string("World!", IOState_tmp1, IOState_tmp2),
 io.nl(IOState_tmp2, IOState_out).
```

mit syntaktischem Zucker (state variable)

```
main(!IO) :-
 io.write_string("Hello, ", !IO),
 io.write_string("World!", !IO),
 io.nl(!IO).
```

Interaktion

```
:- import_module list, string.

main(!IO) :-
   io.read_line_as_string(Result, !IO),
   ( if
       Result = ok(String),
       string.to_int(string.strip(String), N)
     then
       io.format("fib(%d) = %d\n", [i(N), i(fib(N))], !IO)
     else
       io.format("That's not a number...\n", [], !IO)
   ).
```

Schleife und mehrfache Verzweigung

```
main(!IO) :- io.read_line_as_string(Result, !IO), (
    Result = eof,
    io.format("bye bye...\n", [], !IO)
  ;
    Result = ok(String),
    ( if string.to_int(string.strip(String), N)
        then io.format("fib(%d) = %d\n", [i(N), i(fib(N))], !IO)
        else io.format("that isn’t a number\n", [], !IO)
    ),
    main(!IO)
  ;
    Result = error(ErrorCode),
    io.format("%s\n", [s(io.error_message(ErrorCode))], !IO)
  ).
```

Nichtdeterminismus I

```
:- module crypt.
:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is cc_multi.
:- implementation.
:- import_module int, list, string.
```

Nichtdeterminismus II

```
main(!IO) :- io.format("DOG + ANT = CAT\n", [], !IO),
  ( if
      cryptarith(D,O,G,A,N,T,C)
    then
      DOG = 100 * D + 10 * O + G,
      ANT = 100 * A + 10 * N + T,
      CAT = 100 * C + 10 * A + T,
      io.format("%d + %d = %d\n",
                 [i(DOG), i(ANT), i(CAT)], !IO)
    else
      io.format("has no solutions\n", [], !IO)
    ).
```

Nichtdeterminismus III

```
:- pred cryptarith(int::out, int::out, int::out, int::out,
                   int::out, int::out, int::out) is nondet.
cryptarith(D,O,G,A,N,T,C) :-
   Ds0 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], C0 = 0,
   pick( Ds0, G, Ds1),
   pick( Ds1, T, Ds2),
   T = (G + T + C0) mod 10, C1 = (G + T + C0) / 10,
   pick( Ds2, O, Ds3),
   pick( Ds3, N, Ds4),
   A = (O + N + C1) mod 10, A \= 0, C2 = (O + N + C1) / 10,
   pick( Ds4, D, Ds5),
   pick( Ds5, A, Ds6),
   C = (D + A + C2) mod 10, D \= 0, 0 = (D + A + C2) / 10,
   pick( Ds6, C, _).
```

Nichtdeterminismus IV

```
:- pred pick(list(int)::in, int::out, list(int)::out)
          is nondet.
  pick([X | Xs], X, Xs).
  pick([X | Xs], Y, [X | Zs]) :- pick(Xs, Y, Zs).
```

Determinismusarten von Prädikaten

- `det`
  Prädikat trifft auf genau ein Tupel zu
- `semidet`
  Prädikat trifft auf höchstens ein Tupel zu
- `multi`
  Prädikat trifft auf mindestens ein Tupel zu
- `cc_multi` (committed choice multideterministic)
  das Prädikat trifft auf mindestens ein Tupel zu, aber es wird nur ein Tupel betrachtet
- `nondet`
  keine Einschränkung
- `failure`
  schlägt immer fehl

Determinismus abhängig von Übergaberichtung

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
:- mode append(in, in, in) is semidet.

:- func length(list(T)) = int.
:- mode length(in) = out is det.
:- mode length(in(list_skel)) = out is det.
:- mode length(in) = in is semidet.
```

Unterprogramme

- `mode`-Deklaration leitet Unterprogramm von Prädikat ab
- Aussagen in Regelrumpf geeignet umsortiert
- Fehler „nicht belegte Variable“ bereits beim Übersetzen

Ableitung des Determinismus: Konjunktion

Determinismus-Angaben auf Konsistenz geprüft
Und-Verknüpfung

- …kann fehlschlagen, wenn eine Aussage fehlschlagen kann
- …kann Erfolg haben, wenn alle Aussagen Erfolg haben
- …kann mehrmals Erfolg haben, wenn es Erfolg haben kann und eine Aussage mehrmals Erfolg haben kann

Übersetzer: `prime(X), X = 4` → `semidet`
Strenger wäre Modus `fail`

Ableitung des Determinismus: Disjunktion

Oder-Verknüpfung

- …kann fehlschlagen, wenn alle Aussagen fehlschlagen
- …kann Erfolg haben, wenn eine Aussagen Erfolg haben kann
- …kann mehrmals Erfolg haben, wenn mehrere Aussagen Erfolg haben können oder eine Aussage mehrmals Erfolg haben kann
- Sonderfall: Fallunterscheidung

```
:- type ott ---> one ; two ; three.
:- pred p(ott::in, int::out) is det.
p(X, Y) :- ( X=one, Y=1 ; X=two, Y=2 ; X=three, Y=3 ).
:- pred q(ott::in, int::out) is semidet.
q(X, Y) :- ( X = one, Y = 1 ; X = three, Y = 3 ).
```

Variablenbelegungsarten

- `free` – unbelegt
- `ground` – belegt
- `bound` – mit Konstruktor belegt, benötigt Argumente
- `unique` – darf nur in einem Prädikat verwendet werden
- `clobbered` – darf nicht mehr verwendet werden
- `array` – für Felder mit konstanter Zugriffszeit
- `store` – für effiziente zeigerartige Strukturen

Selbstdefinierte Modi

```
:- mode in == (ground >> ground).
:- mode out == (free >> ground).

:- mode di == (unique >> clobbered).
:- mode uo == (free >> unique).
```

Bedarfsauswertung

- Implementiert aber nicht dokumentiert?
- Unter anderem Namen als „partial instantiation“ vorgesehen?

Typen

- Grundtypen: `char` `'a'`, `string` `"abc"`, `int` `42`, `float` `42.0`
- Tupel `{2,'a'}`
- Liste `[2,3,5]`
- Datenverbund

```
:- type playing_card ---> card(rank, suit) ; joker.
:- type rank ---> ace ; two ; three ; four
               ; five ; six ; seven ; eight
               ; nine ; ten ; jack ; queen ; king.
:- type suit ---> clubs ; diamonds ; hearts ; spades.
```

Datenverbund mit benannten Feldern

```
:- type bank account --->
     account( name :: string, no :: int, funds :: float ).

BankAcct = account(Name, AcctNo, Funds),
  ( if Funds >= RequestedSum
      then ... debit RequestedSum from BankAcct ...
      else ... reject debit request ...
  )

  ( if BankAcct^funds >= RequestedSum
      then ... debit RequestedSum from BankAcct ...
      else ... reject debit request ...
  )
```

Automatisch erzeugt Zugriffsfunktionen

Lesezugriff

```
account(A, _, _)^name       = A.
account(_, B, _)^account_no = B.
account(_, _, C)^funds      = C.
```

Schreibzugriff

```
( account(_, B, C)^name       := A ) = account(A, B, C).
( account(A, _, C)^account_no := B ) = account(A, B, C).
( account(A, B, _)^funds      := C ) = account(A, B, C).
```

Schachtelung ist möglich: `record^field^subfield`

Typsynonyme

```
:- type height == float. % In metres.
:- type radius == float. % In metres.
:- type volume == float. % In cubic metres.

:- func volume_of_cylinder(height, radius) = volume.
:- func volume_of_sphere(radius) = volume.
```

- erübrigt Kommentar zur Parameterbedeutung
- unterstützt Regel „primäre Sprachelemente den sekundären Sprachelementen vorziehen“

Polymorphie - Typvariablen

- ```
  :- type tree(T) ---> leaf ;
                       branch(tree(T), T, tree(T)).
  ```
  - `branch(branch(leaf,1,leaf),2,branch(leaf,3,leaf))` hat den Typ `tree(int)`
  - `branch(leaf,{'a',65},branch(leaf,{'b',66},leaf))` hat den Typ `tree({char, int})`
  - `leaf` ist Wert jedes Baumtyps
- `:- type list(T) ---> [] ; [T | list(T)].`
- `:- type maybe(T) ---> no ; yes(T).`

  „had the database community known about `maybe` types they never would have invented `NULL`s and wrecked the relational model“
  ```

Funktionen mit variablen Typen

```
:- func length(list(T)) = int.
length([]) = 0.
length([_ | Xs]) = 1 + length(Xs).
```

Nur für Anschauung – nicht end-rekursiv!

Prädikate mit variablen Typen

```
:- pred search(tree(T)::in, T::in) is semidet.

search(branch(L, X, R), Y) :-
  O = ordering(X, Y),
  ( O = (<), search(R, Y)
  ; O = (=)
  ; O = (>), search(L, Y)
  ).
```

Mercury-Funktion `ordering` festverdrahtet
vergleicht Werte gleichen Typs, liefert `comparison`-Wert:

```
:- type comparison result ---> (<) ; (=) ; (>).
```

Abstrakte Typen

```
:- module dictionary.

:- interface.
:- type dictionary(Key, Value).
:- pred search(dictionary(Key, Value)::in, Key::in,
               Value::out) is semidet.
:- func set(dictionary(Key, Value), Key, Value) =
               dictionary(Key, Value).

:- implementation.
:- import_module list.
:- type dictionary(Key, Value) == list({Key, Value}).
```

Funktionstypen

Funktion höherer Ordnung

```
:- func map(func(T1) = T2, list(T1)) = list(T2).

map(_, []) = [].
map(F, [X | Xs]) = [F(X) | map(F, Xs)].
```

Pädikattypen

Prädikat höherer Ordnung

```
:- pred filter(pred(T), list(T), list(T), list(T) ).
:- mode filter(in(pred(in) is semidet), in, out, out )
                                                   is det.

filter(_, [], [], []).
filter(P, [X | Xs], Ys, Zs) :-
   filter(P, Xs, Ys0, Zs0),
   ( if P(X)
       then Ys = [X | Ys0], Zs = Zs0
       else Ys = Ys0, Zs = [X | Zs0] ).

:- pred filter( pred(T)::in(pred(in) is semidet),
           list(T)::in, list(T)::out, list(T)::out) is det.
```

Polymorphie - Typklassen

```
:- typeclass point(T) where [
     % coords(Point, X, Y):
     % X and Y are the cartesian coordinates of Point
     pred coords(T, float, float),
     mode coords(in, out, out) is det,
     % translate(Point, X_Offset, Y_Offset) = NewPoint:
     % NewPoint is Point translated X_Offset units in the
     % and Y_Offset units in the Y direction
     func translate(T, float, float) = T
     ].
```

Polymorphie - Typbeschränkungen

```
:- pred distance(P1, P2, float) <= (point(P1), point(P2)).
:- mode distance(in, in, out) is det.
distance(A, B, Distance) :-
    coords(A, Xa, Ya),
    coords(B, Xb, Yb),
    XDist = Xa - Xb,
    YDist = Ya - Yb,
    Distance = sqrt(XDist*XDist + YDist*YDist).
```

Geschichte I

- 1993-10: Entwurf des Ausführungsalgorithmus
- 1993-12: Start Entwicklung Mercury-Übersetzer
- 1994-05: Semantische Analyse
- 1994-08: Code-Erzeugung, kurz danach Optimierung
- 1995-02: Mercury-Übersetzer kann sich selbst übersetzen
- 1995-07, 0.3: erste öffentliche Beta-Version
- 1995-09, 0.4: Profiler
- 1996-02, 0.5: Unikat-Prüfer
- 1996-08, 0.6: Funktionale Syntax, Typ- und Übergabemodusableitung
- 1997-08, 0.7: allgemeine Ein- und Ausgabeprädikate, modulübergreifende Optimierung

Geschichte II

- 1997-12: Typklassen
- 1998-04: eigener Debugger
- 1998-05: Fixieren von Prädikaten in Tabellen (tabling)
- 1999-03: Bedarfsauswertung (lazy evaluation)
- 1999-09: Ausnahmebehandlung
- 2000-01: Datenverbund-Syntax
- 2001-02: eigener Codeerzeuger (Assembler statt C)

Geschichte III

- 2002-07: syntaktische Unterstützung für Zustandsvariablen
- 2003-02: Unterstützung für .NET-Codeerzeugung
- 2003-09: verbesserte Speicherverwaltung (garbage collector)
- 2005-03: neue Arten von Aussagen für parallel oder nicht-deterministisch auszuführende Programme
- 2005-05: Funktionale Abhängigkeiten bei Typklassen
- 2006-08: Codeerzeugung in Erlang

Hybridsprache

mwc: „C++ is multiparadigm in the same way
a dog with 4 table legs nailed onto it is an octopus“

„Viel macht viel“ vs. „Weniger ist manchmal mehr“

- kein GOTO → Fortschritt
- keine Seiteneffekte, keine globalen Variablen → Fortschritt
- kein Cut → Fortschritt

Überschneidung der Paradigmen

- Wahrheitswert vs. erfülltes Prädikat (`X = odd(N)`)
- Wahrheitsfunktion vs. Prädikat
- skalare Funktion vs. deterministisches Prädikat
- mengenwertige Funktion vs. nicht-deterministisches Prädikat

Monadkonzept lässt sich schwer von funktionaler Programmierung auf logische Programmierung übertragen.

Beschränkung auf ein Paradigma?

Suche nach einer einfachen Weltformel?
Lässt sich alles mit Logik beschreiben? Mit Mengen? Mit Funktionen?
Lässt sich alles mit Chaostheorie beschreiben? Mit neuronalen Netzen? Mit Fuzzylogik? Mit Quantentheorie? Mit Relativitätstheorie? Mit Raumfahrt? Mit Genetik? Mit Neurologie?