Prolog再考

急がないで推測

オレッグ・キセリョーヴ　　亀山 幸義

日本ソフトウェア科学会第31回大会
2014年9月9日

古典Prologは、項代数、非決定性、単一化, 反例探索、論理と制御の分離 という最も基本的な概念を簡潔にまとめている素敵な言語である。プログラムが双方向に動くのは不思議だ。しかし、現実のPrologプログラムが 持つ問題-カットの多用、算術, FFI, committed choice, 頻繁な発散など -が古典Prologの利点を打消してしまう。
古典Prologは、魅力的な問題だ。古典Prologの勉強をして、その問題から 教訓を学ぶ必要がある。その教訓として、非決定性は基本的だが、標準の 実行モードになるべきではない、そして、遅延推測を使わないと性能が悪 すぎる、ということがあげられる。
古典Prologの利点は、普通の正格な関数型言語で実装できる。本研究では、 遅延推測をOCamlライブラリとして実現し、そのライブラリを使って Prologの典型的な例を記述する。加えて、双方向に動く型推論、 committed choice (maximal munch)を使って双方向に動くparser combinators を記述する。これらは古典Prologで実装できない。これらの実装から、 論理変数の独特な性質、エルブラン領域の列挙の最適化の立場から見た単一化、 WAMへのコンパイルなどが理解できる。

概要

- × 古典Prolog
  - ○ 項代数
  - ○ 非決定性
  - ○ 単一化
  - ○ 反例探索
  - ○ 論理と制御の分離
- ○ 非決定性はそれぞれの好きな関数言語で

Classical Prolog – the archetype and the eponym of logic programming – is a fascinating language, especially for natural language processing and knowledge representation, planning and reasoning. It is greatly appealing to declaratively state the properties of a problem and let the system find the solution. Most intriguing is the ability to run programs ‘forwards’ and ‘backwards’.

We recall these irresistible features below and see why the language that implements them hasn’t turned out as good as it hoped.

古典Prologの全ての利点は、コード2行

append(P,S,L)　prefixとsuffixと全てのリストの関連

```
append([], L, L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

All the best features of Prolog can be illustrated in just two lines of code: the append relation. We should not view what is on the slide as the *code* to append two lists. These two lines define a *relation* between a prefix and a suffix of a list.

古典Prologの全ての利点は、コード2行

append(P,S,L)　prefixとsuffixと全てのリストの関連

```
append([], L, L).
```

- 空リストは各リストのprefixだ
- 各リストは自分のsuffixだ

```
append([H|T],L,[H|R]) :- append(T,L,R).
```

古典Prologの全ての利点は、コード2行

append(P,S,L)　prefixとsuffixと全てのリストの関連

```
append([], L, L).

append([H|T],L,[H|R]) :- append(T,L,R).
```

Tは、Rのprefixながら、cons H Tは、cons H Rのprefix

関連で計算する

?− append([1,2,3],[4,5], X).
⇝ X = [1, 2, 3, 4, 5].

連結関数として

We can then ask: Is there a list X such that append([1,2,3],[4,5],X) holds? Prolog answers Yes, and, furthermore, gives us that list X. As if append were a function to concatenate two lists.

関連で計算する

?− append([1,2],X,[1,2,3,4,5]).

⇝ X = [3, 4, 5].

逆に計算する

Prolog's append is so elegant because it defines a relation among three lists. We specify any two lists and query for the other one that makes the relation hold. For example, let's check if a given list has a given prefix, and if so, remove it. Likewise, we can check for, and remove, a given suffix.

If list concatenation was like running forwards, prefix removal is like running append backwards.

関連で計算する

?− append(X,Y,[1,2,3,4,5]).

⇝

X = [], Y = [1, 2, 3, 4, 5] ;
X = [1], Y = [2, 3, 4, 5] ;
X = [1, 2], Y = [3, 4, 5] ;
X = [1, 2, 3], Y = [4, 5] ;
X = [1, 2, 3, 4], Y = [5] ;
X = [1, 2, 3, 4, 5], Y = [] ;
false .

反連結

Append can also be asked to split a given list in all possible ways, returning its prefixes and suffixes. So, append in Prolog can concatenate, and un-concatenate. If the list is finite, we obtain the finite number of answers.

実際に使われるような例

```
db2([rec(name('山下'),graduated(no),grades([4,4])),
     rec(name('山田'),graduated(yes),grades([2,3,4,4]))]).
```

卒業の学生の記録を削除して、それを印刷し、残したDBを返せよ。 記録がなればそう言いなさい。

The append relation is truly elegant. But let's take a more practical, involved example of using something like append. Suppose we have a database of student records, which contains for each student their name, graduation status and the GPA for each year in attendance. The task is to find a record of a graduated student, print it out and delete. Return the resulting database. If the record is not found, say so.

実際に使われるような例: 削除

```
del_rec ([Rec|LRem],Rec,LRem) :-
    Rec = rec(name(_),graduated(yes),grades(_)).
del_rec ([H|T],Rec,[H|LRem]) :- del_rec(T,Rec,LRem).
```

del_rec(Db,Rec,DbRem)

Db	database
Rec	卒業の学生の記録
DbRem	Recなし Db

This function finds a record of a graduated student and deletes it. It looks quite like append: the second clause has exactly the same structure.

実際に使われるような例: 印刷

```
print_rec (rec(name(N),graduated(yes),grades([G1,G2,G3,G4]))) :-
    print (' Student: '), print (N),print ('   Grades:'),
    print (G1),print (',  '),
    print (G2),print (',  '),
    print (G3),print (',  '),
    print (G4),nl.
```

The printing function is standard.

実際に使われるような例: 組み合わせる

```
prog1(Db) :- del_rec(Db,R,DbRem), print_rec(R), nl, print(DbRem).
prog1(_)  :- print('None graduated').
```

```
?- db2(X), prog1(X).
Student: 山田 Grades:2, 3, 4, 4
```

実際に使われるような例: 問題

```
prog1(Db) :- del_rec(Db,R,DbRem), print_rec(R), nl, print(DbRem).
prog1(_)  :- print('None graduated').
```

```
db3([rec(name('山下'),graduated(no),grades([4,4])),
     rec(name('田中'),graduated(yes),grades([4,4]))]).

?- db3(X), prog1(X).
None graduated
```

非決定性は多過ぎて

But when we use a different database, shown on the slide, the result is totally puzzling. The program said that no student graduated whereas we clearly see the record of the graduated Tanaka. The problem is that Tanaka is very smart and graduated in only two years – and our printing program was unable to deal with it. It failed – which is expected. What is unexpected is that the whole program did not fail, but returned a wrong result. The failure in printing caused the backtracking in the record deletion. We were solving a different problem. What we expected that if the candidate for deletion is found, we should commit to it.

実際に使われるような例: committed choice

```
prog2(Db) :-
  del_rec (Db,R,DbRem) →print_rec (R), nl , print (DbRem);
                          print (' None graduated').
```

```
db3([rec(name('山下'),graduated(no),grades([4,4])),
     rec(name('田中'),graduated(yes),grades([4,4]))]).

?- db3(X), prog2(X).
false.
```

We need so-called committed choice, or soft-cut. Here is how our example looks and works with the soft cut. Now the whole program fails, if the input is unexpected. This is the desired behavior. Predictably, soft-cut (and its rude alternative !/0) are are ubiquitous in practical Prolog programs, occurring almost on every line.

committed choiceについて

```
del([H|T],H,T).
del([H|T],D,[H|R]) :- del(T,D,R).

prog1(L,S) :- (del(L,true,LRem), S= deleted(LRem));
              S= failed.
```

```
?- prog1([false,true,false], S).
~> S = deleted([false, false])   ;
   S = failed .
```

But committed choice is not without problems. To illustrate them, we use a simpler deletion operation.

committed choiceの問題

```
del([H|T],H,T).
del([H|T],D,[H|R]) :- del(T,D,R).

prog3(L,S) :- del(L,true,LRem) →S= deleted(LRem); S= failed.
```

```
?- prog3([true,false,true], S).
⇝  S = deleted([false, true]).

?- prog3(L,S).
⇝  L = [true|_G16], S = deleted(_G16).
```

Committed choice cuts choices. Therefore, when we ‘reverse’ progr3, trying to generate its inputs, we get only one choice, of the list starting with true. But prog3 clearly can delete true from a list even if that element occurs in the middle or at the end of the list.
We can’t really complain: once we cut the choices, we no longer can reverse them.

Constraints immediately come to mind. However, the example is so formulated that we can’t use the standard, in SWI Prolog, constraint-solving libraries. In the original example, the record selection can be arbitrary complex, requiring arbitrarily complex constraints, which generally cannot be solved efficiently. Anyway, Classical Prolog didn’t have constraints.

途中のまとめ

古典Prolog

- ○ 宣言的
- ○ 関連的: 前向き、後ろ向きの実行
- × 非決定性が一般的、決定性が困難、非純粋
- × 型つけない
- × 探索は非柔軟で、よく止らない
- × FFIは純粋を汚染する

Let's review the first part of the talk, good and bad features of Prolog. To emphasize, we are talking about the Classical Prolog. There are various ways of getting around: for example, some Prolog systems have good mode analysis (e.g., Mercury) – but many do not. There are typed Prolog-like systems: Mercury and lambda-Prolog. But they are, unfortunately, much less popular.

The first drawback – non-determinism as the default – is most problematic. Many real-life problems are mostly deterministic, or involve long segments of deterministic computations (e.g., number crunching). Encoding such problems efficiently in Prolog is very difficult, often requiring 'cut' and other impure features.

When a problem suits Prolog, the answer is breathtakingly elegant.

But most of the time it is not.

古典Prologとは

- Kowalski, Clocksin & MellishのProlog
- 論理変数, 単一化, 非決定性
- 汎用プログラム言語として
- database queryやSMT solverのための本の記法はでない

Some say: “Prolog has become modeling language. One of the key features of Prolog is that it is based on first-order logic as a modeling language.” That is not the Prolog that Kowalski, Clocksin and Mellish, and the 5th Generation Project had in mind. The ASP Prolog is just a notation for conveniently expressing sets of constraints, which can be solved in efficient ways. Perhaps this is a better Prolog, but not the one that gripped the imagination in the 70s. Datalog is also Prolog, but it doesn’t even have unification.

I must stress that I do not say non-determinism is bad. It is non-determinism that leads to the elegant formulation of the problem. The problem is not non-determinism, but its amount.

Let's try another approach – instead of designing a language around non-determinism, let's start with a proven language and try to add non-determinism to it.

反省

確立プログラムイング言語 http://okmij.org/ftp/kakuritu/

基本

val dist : (prob $*$ α) list $\rightarrow \alpha$
val fail : unit $\rightarrow \alpha$

val reify0 : (unit $\rightarrow \alpha$) $\rightarrow \alpha$ pV

基本によって表現された便利な関数

val flip : prob $\rightarrow$ bool
val uniformly : α array $\rightarrow \alpha$
...
val exact_reify : (unit $\rightarrow \alpha$) $\rightarrow \alpha$ pV
val reify_part : int option $\rightarrow$ (unit $\rightarrow \alpha$) $\rightarrow$
(Ptypes.prob $*$ α) list
...

An example is a library called Hansei, which adds weighted non-determinism (probabilities) to the ordinary OCaml. We will hush the probabilities in this talk.

The primitives of the library are dist, to non-deterministically choose an element from a list, and fail. There is also a strange sounding function reify0 that turns a program into a tree of choices, letting us program our own search strategies. The library has lots of convenient functions written in terms of primitives, such as flip, the uniform selection, and the exhaustive search through all the choices, which produces the flattened choice tree, or the probability table. The function reify_part is a version of exact_reify. The first argument is the depth search bound (infinite, if None).

反省におけるリスト削除: 紛失なしcommitted choice

```
type bl = Nil | Cons of bool * blist
and blist = unit → bl

let nil : blist = fun () → Nil
let cons : bool → blist → blist = fun h t () → Cons (h,t)
val list_of_blist : blist → bool list

let ftf = cons false @@ cons true @@ cons false @@ nil

let rec del : blist → bool → blist = fun l b →
  match l () with
  | Cons (h,t) → if h = b then t else cons h (del t b)
  | Nil → fail ()
```

We now show how that Hansei can not only delete from a list, with committed choice, but also reverse the deletion without loss.

We first define boolean lists with a non-deterministic spine. Elements could (and should be) be non-deterministic too. We introduce nil and cons as easy-to-use constructors of lists, and a function to convert blists into ordinary OCaml lists so we can show them. The sample list ftf will be used in the examples. The list deletion is defined as an ordinary recursive *function* pattern-matching on the list.

反省におけるリスト削除: 紛失なしcommitted choice

```
type bl = Nil | Cons of bool * blist
and blist = unit → bl

let rec del : blist → bool → blist = fun l b →
  match l () with
  | Cons (h,t) → if h = b then t else cons h (del t b)
  | Nil → fail ()

del ftf true;;
⇝  - : blist = <fun>
```

Executing the deletion by itself does not give an informative answer. Recall that we use the Hansei library to build a probabilistic model, which we then have to run. Running the model determines the set of possible worlds consistent with the probabilistic model. Hansei offers a number of ways to run models and obtain the answers and their weights. We will be using iterative deepening: reify_part. The first argument is the depth search bound (infinite, if None).

反省におけるリスト削除: 紛失なしcommitted choice

```
type bl = Nil | Cons of bool * blist
and blist = unit → bl

let rec del : blist → bool → blist = fun l b →
  match l () with
  | Cons (h,t) → if h = b then t else cons h (del t b)
  | Nil → fail ()

reify_part None @@ fun () →
    list_of_blist @@ del ftf true
⇝
    [( 1. , [false ; false ])]
```

Running the del model gives the expected result. We have defined del as a function, and can indeed run it as the list deletion function, 'forwards'. The answer is returned with the probability 1 – del is deterministic. (Actually, del can fail: it is a partial, or semi-deterministic function.)

論理変数でしょうか

```
reify_part    None @@ fun () →
    let  l  = fun ()  → Cons (flip   0.5,  nil )  in
    (list_of_blist      l ,  list_of_blist      @@ del l  true )
⇝
  [(0.25,  ([  false  ],  []));  (0.25,  ([ true ],  []))]
```

But how can we run the deterministic function like del backwards? The first attempt is just to run it on a randomly generated list. In Hansei, a boolean X is modeled as a non-deterministic boolean `flip 0.5`. But something is wrong with the Hansei code!

論理変数

```
val letlazy : (unit → α) → (unit → α)

reify_part None @@ fun () →
    let l = letlazy @@ fun () → Cons (flip 0.5, nil ) in
    (list_of_blist l , list_of_blist @@ del l true )
⇝ [(0.5, ([ true ], []))]
```

- 呼びなら選択
- 波関数の崩壊

We need the magical function *letlazy*, which at first blush looks like an identity function. It is another primitive of Hansei. It takes a thunk and returns a thunk. When we force that thunk, we force the original one, *and* remember the result. All further forcing return the same result. In functional-logic programming, this is called “call-time choice”. In quantum mechanics, in is called “wavefunction collapse”. Before we observe a system, for example, a still spinning coin, there could indeed be several choices for the result. After we observed the system, all further observations give the same result.

Now the code gives the expected answer.

論理変数: memoされた発生器

```
let rec a_blist () =
  letlazy (fun () →
    dist [(0.5, Nil); (0.5, Cons(flip 0.5, a_blist ()))])

reify_part (Some 5) @@ fun () →
    let l = a_blist () in
    (list_of_blist l, list_of_blist @@ del l true)

⇝ [(0.03125, ([ false ; true], [false ]));
   (0.125, ([true], []));
   (0.03125, ([true; false ], [false ]));
   (0.03125, ([true; true], [true]))]
```

To invert del, we run it on *any* list. We define a generator for any blists (with letlazy) and use it as a logic variable `X`. In Prolog, our deterministic deletion, with soft cut, was stuck on generating all lists with true at the head. In Hansei, we can generate other lists.

論理変数: memoされた発生器

```
let rec a_blist () =
  letlazy (fun () →
    dist [(0.5, Nil); (0.5, Cons(flip 0.5, a_blist ()))])

reify_part None @@ fun () →
    let l = a_blist () in
    list_haslen l 3;
    (list_of_blist l, list_of_blist @@ del l true)
⇝ [(0.0078, ([false; false; true], [false; false]));
   (0.0078, ([false; true; false], [false; false]));
   (0.0078, ([false; true; true], [false; true]));
   (0.0078, ([true; false; false], [false; false]));
   (0.0078, ([true; false; true], [false; true]));
   (0.0078, ([true; true; false], [true; false]));
   (0.0078, ([true; true; true], [true; true]))]
```

We can do better! We run without the upper bound on the search, and it terminates! We get all possible lists of three elements, with the exception of the list with all false, where del true fails. We can recover from the failure: see the accompanying code for soft-cut.

無先行のリスト

```
reify_part  (Some 5) @@ fun () →
    let  l  = a_blist  ()  in
    (list_of_blist     l ,  list_of_blist     @@ del l  true )
```

```
let  rec  list_haslen   :  blist  → int  → unit  = fun l  n →
  match (l  (), n)  with
  |  (Nil ,0)                          → ()
  |  (Cons (_, t ), n)  when n >0 → list_haslen   t  (n−1)
  |  _                                 → fail  ()
```

list_haslen l 3には たかが4つthunkが強制させる

そのlazyじゃない

- OCamlのlazyじゃない
- Schemeのdelayじゃない
- Haskellの遅延評価じゃない

その遅延評価は全体のstoreを変化させる

非決定性のためは、地元の第一級storeが必要

We have seen the crucial role of laziness, to delay the computation and memoize its result. OCaml has a facility to delay a computation and memoize the result – called lazy. Scheme has delay. In Haskell (GHC), lazy evaluation is pervasive. None of them do what we want. They are all implemented via mutation of the ordinary, or global, or shared memory – shared across all possible worlds, resulting from a choice. It is useful to think of a non-deterministic choice, flipping a coin, as splitting the current world. In one world, the coin came up 'head', in the other it came 'tail'. If we are to memoize, cache the result, we should use different memo tables for different worlds, because different worlds have different choices.

In short, non-deterministic laziness needs first-class memory – which is what Hansei implements.

Hansei: Summary

- 宣言的
- 関連のふりした
- 決定性が一般的,
  非決定性が易しい
- 型付け
- 探索がプログラムできる
- FFIが易しい

http://okmij.org/ftp/kakuritu/logic-programming.html

まとめ

古典Prologは is the exquisite square peg in the world with mostly round holes

「本物論理プロ グラ ミン グは各方の好きな関数言語へ」という例

The concise and declarative formulation of problems is the gift of non-determinism and the reason for its invention. Classical Prolog makes non-determinism the default computational mode. Taken to such extreme, non-determinism turns from virtue into vice. Quite many computations and models are mostly deterministic. Implementing them in Prolog with any acceptable performance requires the extensive use of problematic features such as cut. Purity is also compromised when interfacing with mainstream language libraries, which are deterministic and cannot run backwards. Divergence is the constant threat, forcing the Prolog programmers to forsake the declarative specification and program directly against the search strategy. All in all, Classical Prolog is the exquisite square peg in the world with mostly round holes.

We have seen the standard logic programming examples in the completely standard OCaml. We did not implement Prolog, Kanren, Curry in OCaml. Rather, we used OCaml directly, as it is. In particular, we call any OCaml function from any OCaml library directly, and can be called by it. You can probably do the similar non-deterministic programming in your language.

The talk is meant to be provocative, so if it hasn't been so far, I'll end it on the provocative note: everyone should learn and know Classical Prolog, and nobody should use it.

まとめ

古典Prologは is the exquisite square peg in the world with mostly round holes

「本物論理プロ グラ ミン グは各方の好きな関数言語へ」という例

Prologを学べ、でも実際に使うな

The concise and declarative formulation of problems is the gift of non-determinism and the reason for its invention. Classical Prolog makes non-determinism the default computational mode. Taken to such extreme, non-determinism turns from virtue into vice. Quite many computations and models are mostly deterministic. Implementing them in Prolog with any acceptable performance requires the extensive use of problematic features such as cut. Purity is also compromised when interfacing with mainstream language libraries, which are deterministic and cannot run backwards. Divergence is the constant threat, forcing the Prolog programmers to forsake the declarative specification and program directly against the search strategy. All in all, Classical Prolog is the exquisite square peg in the world with mostly round holes.

We have seen the standard logic programming examples in the completely standard OCaml. We did not implement Prolog, Kanren, Curry in OCaml. Rather, we used OCaml directly, as it is. In particular, we call any OCaml function from any OCaml library directly, and can be called by it. You can probably do the similar non-deterministic programming in your language.

The talk is meant to be provocative, so if it hasn't been so far, I'll end it on the provocative note: everyone should learn and know Classical Prolog, and nobody should use it.

A historical note

"One can see now how this talk in 1957 must have motivated Gilmore, Davis and Putnam to write their Herbrand-based proof procedure programs. Their papers ... were based fundamentally on the idea of systematically enumerating the Herbrand Universe of a proposed theorem – namely, the (usually infinite) set of all terms constructible from the function symbols and individual constants which (after its Skolemization) the proposed theorem contained. This technique is actually the computational version of Herbrand's so-called Property B method. ... These first implementations of the Herbrand FOL proof procedure thus revealed the importance of trying to do better than merely hoping for the best as the exhaustive enumeration for only ground on, or than guessing the instantiations that might be the crucial ones in terminating the process. In fact, Herbrand himself had already in 1930 shown how to avoid this enumerative procedure, in what he called the Property A method. The key to Herbrand's Property A method is the idea of unification.

A historical note (cont)

Herbrand's writing style in his doctoral thesis was not, to put it mildly, always clear. As a consequence, his exposition of the Property A method is hard to follow, and is in fact easily overlooked. At any rate, it seems to have attracted no attention except in retrospect, after the independent discovery of unification by Prawitz thirty years later....

Once I had managed to recast the unification algorithm into a suitable form, I found a way to combine the Cut Rule with unification so as to produce a rule of inference of a new machine-oriented kind. It was machine-oriented because in order to obtain the much greater deductive power than had hitherto been the norm, it required much more computational effort to apply it than traditional human-oriented rules typically required. In writing this work up for publication, when I needed to think of a name for my new rule, I decided to call it "resolution", but at this distance in time I have forgotten why. This was in 1963."

John Alan Robinson: "Computational Logic: Memories of the

And of course the resolution is the foundation of Prolog. What I want to emphasize is that the unification was invented to cope with a large or infinite search space. It is an optimization. But so is laziness...