Non-deterministic choice in a conventional programming language

Enough for logic programming?

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

National Institute of Informatics, Japan
December 18, 2012

Outline

- **Introduction**

Prolog

Hansei

Reversible parsers, Zebra

Type inference

Conclusions

Logic programming is a fascinating approach, especially for AI and natural language processing. It is greatly appealing to declaratively state the properties of the problem and let the system find the solution. Most intriguing is the ability to run programs 'forwards' and 'backwards'.

However, the built-in search methods of logic programming systems don't fit all problems and hardly if at all customizable. Mainly, quite many computations and models are mostly deterministic. Implementing them in a logic programming language is significantly inefficient and requires extensive use of problematic features such as cut. Another problem is interfacing logic programs with mainstream language libraries: if mode analysis is not available (as is often the case), one has to live with run-time instantiatedness errors.

An alternative to logic programming, where non-determinism is default, is a deterministic programming system (such as Scheme, OCaml, Scala or Haskell – or even C) with (probabilistic) non-determinism as an option. Is this a good alternative? We explore this question. We will use Hansei – a probabilistic programming system implemented as a library in OCaml – to solve a number of classic logic programming problems, from zebra to scheduling, to parser combinators, to reversible type checking.

Outline

Pure Prolog

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

First-order theory of append

Since the talk is about alternatives to Prolog, we ought to recall the golden standard: Prolog. All the best features of Prolog can be illustrated in only two lines of code: the append relation.
The two lines of code on the slide define the first-order theory for append:

- the 3-place predicate append,
- an axiom: forall L. append([],L,L) holds.
- a rule: forall H T L L2. whenever append(T,L,L2) holds, append([H|T],L,[H|L2]) also holds. That is, we can add an element to T and L2.

Pure Prolog

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

?− append([t,t, t],[ f , f], X).
        X = [t,  t,  t,  f,  f].
```

append as a concatenation function

Is there a list X such that append([t,t,t],[f,f],X) holds? Prolog answers Yes, and, furthermore, gives us that list X. As if append were a function to concatenate two lists.

Pure Prolog

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

?− append([t,t], X,[t, t, t, f, f]).
     X = [t,  f,  f].
```

concatenating backwards

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.

Pure Prolog

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

?− append([t,t, t], X,R).
      R = [t,  t,  t| X].
```

concatenating in another way

There are more ways to run append; for example: find all lists R with a given prefix [t,t,t] and an arbitrary suffix X. The answer is given on one line, which, however, compactly represents an infinite number of solutions. Hence a question in Prolog may have more than one answer. We get the first hint of non-determinism.

The compact representation is less wonderful than it looks. For example, Prolog cannot compactly represent all *boolean* lists with a given prefix.

Pure Prolog

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

 ?− append(X,[f,f], R).
       R = [f,  f]  ;
       R = [_G328, f,  f]  ;
       R = [_G328, _G334, f,  f]  ;
       R = [_G328, _G334, _G340, f,  f].
```

and un-concatenating

If we ask for all lists with the [f,f] suffix, Prolog lists the solutions, as an infinite stream. Non-determinism becomes clear.

Pure Prolog

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

```
?- append(X,Y,[t,t,t,f,f]).
     X = [],    Y = [t, t, t, f, f] ;
     X = [t],   Y = [t, t, f, f] ;
     X = [t, t],   Y = [t, f, f] ;
     X = [t, t, t],   Y = [f, f] ;
     X = [t, t, t, f],   Y = [f] ;
     X = [t, t, t, f, f],   Y = [] ;
     false .
```

another un-concatenation: splitting

Append can also split a given list in all possible ways, returning its prefixes and suffixes. If the list is finite, we obtain the finite number of answers.

FFI

```
lenf ([],0).
lenf ([ _| T],N) :- lenf(T,N1), N is  N1 + 1.
```

Append is truly the best illustrative example of Prolog, showing off building and unifying terms. Once we venture beyond the term algebra, things easily turn awry. Arithmetic in Prolog is already "outside" and has to be accessed using a sort of an FFI. Alas, most of such 'foreign' libraries are deterministic, functional rather than relational.

As an example, let's define lenf that relates a list and its length, an integer.

FFI

```
lenf ([],0).
lenf ([_| T],N) :- lenf(T,N1), N is  N1 + 1.

?- lenf([1,2,3],   N).
     N = 3.
```

We easily find the length of a given list.

FFI

```
lenf ([],0).
lenf ([ _| T],N) :- lenf (T,N1), N is  N1 + 1.

?- lenf(X,3).
   X = [_G280, _G283, _G286] ;
   <divergence>
```

We can also fix the length (for example, 3) and ask for all lists of that length. After printing the first solution, Prolog indicates that there may be more. If we want to see them, Prolog goes into an infinite loop. The reader is encouraged to find out the cause of the divergence.

FFI

```
lenf ([],0).
lenf ([ _| T],N) :- lenf(T,N1), N is  N1 + 1.

?- lenf([1,2],1).
    false .
```

Here we test if a given list has a given size.

FFI

```
lenf ([],0).
lenf ([_| T],N) :- lenf(T,N1), N is  N1 + 1.

?- lenf([1,2 | X],1).
    <divergence>
```

Yet if we change the example slightly (asking if a list with the prefix [1,2] has the length 1) we get no answer. The interpreter loops. Why? Here's a hint: a common problem is conjoining a generator that keeps generating with a test that keeps rejecting.

FFI

```
lenf ([],0).
lenf ([_| T],N) :- lenf(T,N1), N is N1 + 1.

lenf1 ([],0).
lenf1 ([_| T],N) :- N is N1 + 1, lenf1(T,N1).

?- lenf1([1,2,3], N).
ERROR: user://4:78:
is/2: Arguments are not sufficiently instantiated
```

Perhaps moving the test before the generator would help? (Prolog, or the SLD resolution, evaluates goals in a conjunction strictly left-to-right.) The divergence is certainly cured, replaced by an error. The error tells us that the built-in arithmetic of Prolog is not relational. We cannot run addition 'backwards'.

FFI

```
lenf ([],0).
lenf ([_| T],N) :- lenf(T,N1), N is N1 + 1.

lenf1 ([],0).
lenf1 ([_| T],N) :- N is N1 + 1, lenf1(T,N1).

?- lenf1([1,2,3], N).
ERROR: user://4:78:
is /2: Arguments are not sufficiently instantiated
```

Arithmetic can be relational

- Constraint solving
- Arithmetic in pure Prolog (FLOPS 2008)

To be sure, arithmetic can be relational. Many Prolog systems, for example, SWI Prolog, have arithmetic constraint solvers. Relational arithmetic can also be implemented in Pure Prolog, see our FLOPS 2008 paper.

Pure Prolog: Summary

- ◯ Declarative: find a solution by stating a problem
- ◯ Relational: running forwards and backwards
- ✕ Non-determinism is default,
  determinism is difficult and impure
- ✕ Untyped
- ✕ Search is too rigid, divergence is too common
- ✕ FFI breaks purity, causes instantiatedness errors

Let's review the first part of the talk, good and bad features of Prolog. To emphasize, we are talking about the pure 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.

Outline

Are there alternatives? Let's try adding non-determinism to an ordinary language, where determinism is default.

Hansei

`http://okmij.org/ftp/kakuritu/`

Basic functions

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

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

Convenient derived functions

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).

Append in Hansei

```
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 t3 = cons true (cons true (cons true nil ))
let f2 = cons false (cons false nil )

let rec append l1 l2 =
  match l1 () with
  | Nil          → l2
  | Cons (h,t) → cons h (fun () → append t l2 ())
```

Recall, the append relation is the best illustration of Prolog. Let's see if we can represent it in Hansei. 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. Sample lists t3 and f2 will be used in the examples. The append is defined as an ordinary recursive *function* pattern-matching on the list.

Append in Hansei

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

let rec append l1 l2 =
  match l1 () with
  | Nil          → l2
  | Cons (h,t) → cons h (fun () → append t l2 ())

append t3 f2;;
  - : blist = <fun>
```

Executing the append 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: the model of the model. The set of model outputs in these worlds is the set of answers. 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).

Append in Hansei

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

let rec append l1 l2 =
  match l1 () with
  | Nil          → l2
  | Cons (h,t) → cons h (fun () → append t l2 ())

reify_part  None (fun () →
  list_of_blist    (append t3 f2))

    [(1.,  [true; true; true; false ; false ])]
```

Running the append model gives the expected result. We have defined append as a function, and can indeed run it as the concatenation function, ‘forwards’.

Logic variables?

Prolog

```
?- bool(X), append([X],[f,f], R).
X = t ,
R = [t, f, f] ;
X = f ,
R = [f, f, f].
```

Hansei

```
reify_part (None (fun () →
  let l = fun () → Cons (flip 0.5, nil) in
  list_of_blist (l, list_of_blist (append l f2))

[(0.25, ([false], [false; false; false]));
 (0.25, ([false], [true; false; false]));
 (0.25, ([true], [false; false; false]));
 (0.25, ([true], [true; false; false]))]
```

Prolog could run append forwards in several ways, for example, to determine lists with [f,f] as the suffix and a one-element prefix. The slide shows the Prolog code and the corresponding Hansei code. In Hansei, a boolean X is modeled as a non-deterministic boolean `flip 0.5`. But something is wrong with the Hansei code!

Logic variables

```
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 (append l f2))
```

```
[(0.5, ([ false ], [ false ; false ; false ]));
(0.5, ([ true ], [ true ; false ; false ]))]
```

- call-time choice
- wave-function collapse

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.

Logic variables: memoized generators

```
?- append([t,t,t], X,R), boollist (X), boollist (R).
```

Let's recall another Prolog example, enumerating all boolean lists with [t,t,t] as the prefix. Unlike earlier Prolog code, we now make sure the lists are really boolean, whose elements are only t or f.

Logic variables: memoized generators

```
?- append([t,t,t], X,R), boollist (X), boollist (R).

X = [],
R = [t, t, t] ;
X = [t],
R = [t, t, t, t] ;
X = [t, t],
R = [t, t, t, t, t] ;
X = [t, t, t],
R = [t, t, t, t, t, t] ;
X = [t, t, t, t],
R = [t, t, t, t, t, t, t] ;
X = [t, t, t, t, t],
R = [t, t, t, t, t, t, t, t]   ...
```

Where is [t, t, t, f]?

Prolog indeed gives an infinite stream of answers. However, it does seem to be stuck on `t`. For example, [t,t,t,f] is also a list with [t,t,t] as the prefix, but we won't see it. The built-in search strategy of Prolog is incomplete.

Logic variables: memoized generators

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

?− append([t,t, t], X,R), boollist (X), boollist (R).


reify_part (Some 3) (fun() →
  let x = a_blist () in
  list_of_blist (append t3 x))

 [(0.5, [true; true; true]);
  (0.125, [true; true; true; false ]);
  (0.125, [true; true; true; true])]
```

Here is the same example in Hansei. We define a generator for blists (with letlazy) and use it as a logic variable `X`. We are no longer stuck on generating all true lists.

List comparison

```
let rec bl_compare l1 l2 =
  match (l1 (), l2 ()) with
  | (Nil , Nil ) → true
  | (Cons (h1,t1), Cons (h2,t2)) → h1 = h2 && bl_compare t1 t2
  | _            → false
```

Prolog can also run append backwards. Can Hansei? Let's first define the comparison function on blist, in the straightforward way.

Un-appending

```
?− append([t,t, t],[ f , f ], L),  append([t, t ], X,L).

reify_part    None (fun() →
   let  l  = append t3 f2 in
   let  x = a_blist   ()  in
   let  r  =  append (cons true  (cons true  nil ))  x   in
   if    not  (bl_compare r  l )   then fail   ();
   list_of_blist      x)

[(0.0078125,  [true ;  false  ;  false ])]
```

Running backwards ≡ generate-and-test

How come it terminated?

The principle for running a function backwards is generate-and-test. We generate all possible lists x, prepend [true,true] and check the result matches l. We effectively un-concatenate l.

In Prolog, conjoining a generator for an infinite stream with a test often leads to divergence. The generator keeps producing and the test keeps rejecting. How come Hansei code terminated, with *no* upper bound on the search depth? Hansei search space is finite?

Splitting a list

```
?− append(X,Y,[t,t,t,f,f]).

reify_part  None (fun() →
  let l = append t3 f2 in
  let x = a_blist () in
  let y = a_blist () in
  let r = append x y in
  if not (bl_compare r l) then fail ();
  (list_of_blist x, list_of_blist y)

[(0.000244140625, ([], [true; true; true; false; false ]));
 (0.000244140625, ([true], [true; true; false; false ]));
 (0.000244140625, ([true; true], [true; false; false ]));
 (0.000244140625, ([true; true; true], [false; false ]));
 (0.000244140625, ([true; true; true; false ], [false ]));
 (0.000244140625, ([true; true; true; false; false ], []))]
```

How come it terminated?

Our last Prolog example demonstrated how the append relation splits the list, in all possible ways. The top of the slide recalls that Prolog code. Underneath is Hansei code.

Laziness principle

- How to guess
- How to go proceed as if we guessed
- Delay the actual guess till the latest possible moment hoping that moment never arrives

Let me stress the point of the talk: how to write program that guess, how to guess intelligently, how to proceed as if we guessed (and delay the actual guess until more information becomes available, counting that if some other guesses are wrong, the delayed guess is not worth making).

Not *that* laziness

- Not `lazy` of OCaml
- Not `delay` of Scheme
- Not laziness of Haskell

That laziness mutates global (shared) memory

- mutation affects all possible worlds
- `letlazy` memoizes different results in different possible worlds
- `letlazy` needs world-local memory
- remember `fork ()` and Unix processes?

Non-deterministic laziness needs first-class memory

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.

Outline

I have many more examples of classical logic programming puzzles written in Hansei – for example, the zebra puzzle. I have non-classical examples – reversible parser combinators. Please ask me.

Outline

As a final example, I'll show invertible type-checking.

Type inference: terms

Simply typed λ-calculus with integer literals

```
type varname = string
type term_v =
  | I of int | V of varname
  | L of varname * term | A of term * term
and term = unit -> term_v

let (%) e1 e2 = fun() -> A (e1,e2)
let lam v t   = fun() -> L (v, t) and num x = fun() ->I x

let a_term () : term =
  let var () = "x" ^ string_of_int (geometric 0.1) in
  let rec loop () =
    dist [(0.1, fun () -> I 1);
          (0.1, fun () -> V (var ()));
          (0.4, fun () -> L (var (), letlazy loop));
          (0.4, fun () -> A (letlazy loop, letlazy loop))] ()
  in letlazy loop
```

Our language is simply-typed lambda-calculus with integer literals. The slide shows the syntax of terms (literals, variables, abstraction and application), and sugar functions for building terms conveniently. Next we define a generator for terms. The sequence of terms is infinite – as is the sequence of variable names.

Type inference: types

```
type base_t = Int
type tp_v = B of base_t | Arr of tp * tp
and tp = tp_v option -> tp_v

let int : tp = tp_pure (B Int)
let arr : tp -> tp -> tp = fun t1 t2 -> tp_pure (Arr (t1, t2))


let a_tp : tp =
  let a_baset () = Int in
  let rec loop = function
    | None ->dist [ (0.5, B (a_baset ()));
                    (0.5, Arr (tp_memo loop, tp_memo loop))]
    | Some tv ->tv
  in loop
```

Types are base types (Int) and arrow types. We show the sugar functions and the generator.

Type inference: forwards and backwards

```
let rec typeof : gamma →term →tp = fun gamma exp →
 match exp () with
  | I _      → int
  | V name →begin try List.assoc name gamma
                      with Not_found → fail ()
                end
  | L (v, body) →
      let targ = new_tvar() in
      let tbody = typeof ((v, targ) :: gamma) body in
      arr targ tbody
   | A (e1,e2) →
       let tres = new_tvar() in
       tp_same (typeof gamma e1) (arr (typeof gamma e2) tres);
       tres
```

A type for a term, terms for a type, or all well-typed terms
This is OCaml, not Prolog or Curry!

The type-inference code fits on a single slide. The code looks quite like the familiar typing rules. We can either obtain the type for a term, or generate all terms for a given type or all well-typed terms. [show demo live]

Outline

Hansei: Summary

- Declarative: find a solution by stating a problem
- Fake Relational: running forwards and backwards
- Determinism is default,
  non-determinism is rather easy
- Typed
- Search is programmable
- FFI is native

Conclusions

Do guess ... but not a moment too soon

- Guess
- Delay a guess until the last moment
- Fail sooner
- Test-and-generate (rather than generate-and-test)

Logic programming in *your* language

- Standard logic programming – in the standard OCaml
- Rather than implementing Prolog or Curry in OCaml
- Calling any OCaml function, with no FFI

The principles naturally lead to the constraint (logic) programming. 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.

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...