Prolog再考: 急がないで推測

オレッグ・キセリョーヴ　亀山 幸義 .

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。

1 Introduction

Classical Prolog [1, 2] – the archetype and the eponym of logic programming – is a fascinating language, especially for natural language processing [3, 4] and knowledge representation, planning and reasoning [5, 6]. 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 in §2. 1.

The concise and declarative formulation of problems is the gift of non-determinism and the reason for its invention [7]. 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.

The case in point is the ubiquitous committed choice [8], which is necessary to express the pervasive 'maximal munch' parsing convention (§5) as well as the 'don't care non-determinism'. For these reasons, committed choice is natively supported in Prolog, as the 'soft-cut'. However, Prolog programs with committed choice can no longer be run backwards: see §5. 1.

The history of Prolog [2], designed on the foundation of non-determinism and resolution, is adapting, restricting or overcoming these ideas to make

Rethinking Prolog
Oleg Kiselyov, Yukiyoshi Kameyama, Tsukuba.

the language general-purpose. An alternative is to start with a mature general-purpose, deterministic programming language, with a proven record of solving real-world problems – and add non-determinism. Is this a good alternative? We explore this question in §3. We use Hansei – a probabilistic programming system implemented as a library in OCaml [9, 10] – to solve a number of classic logic programming problems, from zebra to scheduling, to parser combinators, to reversible type checking. The complete code accompanying the is available at `http://okmij.org/ftp/kakuritu/logic-programming.html`.

Many mature functional languages easily let non-determinism in, thanks to the features like MonadPlus in Haskell or delimited control libraries in Scala, OCaml or Scheme. Alas, the cheaply added non-determinism has a ridiculously poor performance even on toy problems. Making non-determinism usable requires non-trivial insight: lazy sharing, see §3. 2.

As a larger case study, §5. 2 presents a parser combinator library to build maximal-munch parsers that are reversible: a parser may run forwards to parse a given string, and backwards to generate all parseable strings, the language of its grammar. Such reversible parser combinators with the maximal munch cannot be idiomatically implemented in Classical Prolog.

Our argument is the argument for functional logic programming [11] – however, realized not as a standalone language such as Curry but as a library in the ordinary programming language. We stress that we do not advocate the embedding of Prolog in a general-purpose language. Many such embeddings have been done (in Scheme, Haskell, Scala, etc), all sharing the drawbacks of Classical Prolog. Rather, we advocate transcending Prolog: taking its best features – separation of the model specification from the search and non-determinism – and bringing them into the conventional functional-programming language. Such bottom-up approach is not only practical but also theoretically revealing. We see in §6 how logic variables and unification naturally emerge as a "mere optimization" of non-deterministic search.

2 Fascination and Disappointment of Classical Prolog

In this section we recall how Classical Prolog continues to hold our fascination. We also recall the disappointments and eventual realization that Classical Prolog is in reality not a general-purpose programming language. This realization drives us to introduce the best Prolog features in the general purpose, functional languages, in §3.

2. 1 The Append example

All the best features of Prolog can be illustrated in only two lines of code: the append relation:

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

The three-place predicate append establishes the relation between three lists l1, l2 and l3 such that l1 is a prefix and l2 is the corresponding suffix of l3. The two lines declare that the empty list is a prefix of any list, a list is a suffix of itself, and a list prefix is the sequence of its initial elements. When we ask a Prolog system if there is a list X such that append([t,t,t],[f,f],X) holds, Prolog answers 'Yes'. Furthermore, it gives us that list X – as if append were a function to concatenate two lists.

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

Prolog's append is however is not just a function: it is a relation. We may specify any two lists and query for the other one that makes the relation hold. For example, let us check if a given list has a given prefix, and if so, remove it (that is, obtain

the corresponding suffix).

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

Likewise, we can check for, and remove, a given suffix. If the list concatenation was like running append forwards, prefix removal is like running it backwards.

There are more ways to run append; for example: find all lists R with the given prefix [t,t,t] and an arbitrary suffix X.

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

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

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

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.

```
?− 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 .  % no more answers
```

2.2 Disappointments

The append relation is the best illustration of Prolog – of its fascination, and, as we see in this section, of some of its disappointments. Recall our example of finding all lists R with the given prefix [t,t,t].

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

The given answer compactly represents the infinite set of lists. Only some of them are *boolean* lists, that is, made of elements t and f. We cannot enforce the type of the list elements through a static type system: Classical Prolog is untyped. One may think the lack of a type system is a minor drawback. The easy-to-write specification for boolean lists:

```
bool(t).  bool(f).
boollist ([]).
boollist ([H|T]) :− bool(H), boollist (T).
```

lets us declare that the lists R and X in the original example are in fact boolean:

```
?− 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] ;
  ...
```

The result is disappointing. First, boolean lists with the given prefix are no longer compactly represented. More worrisome, Prolog is stuck on t. For example, [t,t,t,f] is also a boolean list with the prefix [t,t,t], but we do not see it among the answers. The built-in search strategy of Prolog is incomplete. If we change the order of the predicates in the conjunction

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

we find to our dismay that Prolog loops after giving the first solution. Therefore, Classical Prolog

programs are not as declarative as one may think: the order of predicates and clauses matters a great deal. One must be very familiar with the evaluation strategy to write programs that produce any result, let alone produce the result fast.

There are more problems, such as numerical calculations or interfacing with foreign functions. They can be dealt with various success in modern Prolog systems, via mode inference and constraint-solving systems – which take us beyond Classical Prolog.

The biggest problem is non-determinism as the default. 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, which destroy the reversibility and do not play well with constraint solving. §4 gives one example, of the need for restricting non-determinism and the problem it causes for Prolog.

When a problem suits Prolog, the answer is breathtakingly elegant. But most of the time it is not.

3 An alternative to Prolog

As an alternative to Classical Prolog we add non-determinism to an ordinary language, where determinism is default. An example is a library called Hansei[†1], which adds weighted non-determinism (probabilities) to the ordinary OCaml. (We will ignore the probabilities in this paper.)

The primitives of the library, see Figure 1, 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 αpV, letting us program our own search strategies. The library has many convenient functions written in terms of the primitives,

†1 http://okmij.org/ftp/kakuritu/

Basic functions

```
type prob = float
val dist     : (prob * α) list  → α
val fail     : unit  → α
val reify0   : (unit  → α) → α pV
val letlazy  : (unit  → α) → (unit  → α)
```

Convenient derived functions

```
val flip          : prob → bool
val uniformly     : α array  → α
...
val exact_reify   : (unit  → α) → α pV
val reify_part    : int  option → (unit  → α) →
                    (prob * α) list
...
```

図 1 Hansei interface

such as flip, flipping a coin, and the uniform selection; exact_reify exhaustively searches through all the choices and produces the flattened choice tree, or the probability table.

Hansei (and similar libraries for Scala or Haskell) show that just adding non-determinism to an established language is straightforward. With little effort we can already write Prolog-like programs, and we do in §3. 1. We also see that the cheap non-determinism is cheap indeed: it performs poorly and is prone to divergence. §3. 2 presents a smart alternative. Our running example is writing the classic append relation of Prolog in Hansei.

3. 1 Cheap non-determinism

At first blush, there is little to do. OCaml already has a built-in operation @ to concatenate two lists. Extending this function to a relation is also straightforward, thanks to non-determinism. We demonstrate on the example of running append backwards: un-concatenating a given list and producing all its possible prefixes and the corresponding suffixes. The task thus is to represent the following Prolog code in OCaml

```
?− append(X,Y,[t,t,t,f,f]).
```

The key idea is that running backwards is tantamount to generate-and-test: in our case, generating candidate prefixes and suffixes and then testing if they make up the given list. We merely need a generator of lists, all boolean lists in our example.

```
let rec a_list () =
  if flip 0.5 then []
    else flip 0.5 :: a_list ()
```

Declaratively, a boolean list is either [] or a boolean list with either **true** or **false** at the head. In Hansei terms, the thunk a_list is 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 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, a version of exact_reify whose first argument is the depth search bound (infinite, if None). For example, we test a_list by generating a few sample boolean lists:

```
reify_part (Some 3) a_list
⇝ [(0.5, []); (0.125, [false]); (0.125, [true])]
```

Everything is set to implement our idea of running @ backwards, to unconcatenate the sample list t3f2 obtaining all its prefixes and the corresponding suffixes.

```
let t3f2 = [true; true; true; false; false]
reify_part (Some 25) (fun() →
  let x = a_list () in
  let y = a_list () in
  let r = x @ y in
  if not (r = t3f2) then fail ();
  (x, y))
⇝
[(0.0002, ([], [true; true; true; false; false]));
 (0.0002, ([true], [true; true; false; false]));
 (0.0002, ([true; true], [true; false; false]));
 (0.0002, ([true; true; true], [false; false]));
 (0.0002, ([true; true; true; false], [false]));
 (0.0002, ([true; true; true; false; false], []))]
```

It really works as intended, although it takes about 2 seconds even for such a trivial example. Alas, if we increase the search bound (the first argument of reify_part) to 35, the program diverges. It is not difficult to see why: a_list really generates all possible boolean lists; only very few of them add up to t3f2; the others will have to be rejected. The problem with cheap non-determinism is generating vastly too many candidate solutions, almost all of which are rejected.

3.2 Smart non-determinism

To use non-determinism effectively requires sophistication: to avoid generating and considering clearly failing solutions. The key idea is laziness – delaying the choices till the last possible moment. Looking back at Prolog gives us a hint. OCaml lists are fully determined: [**true**; **false**] is the list of the definite size with definite elements. Prolog lets us write partly determined lists, such as [t|X]; we know the head of the list but do not yet know what follows. Comparing this list with others, such as [t,f|Y], increases our knowledge. Some other comparisons, e.g., with [f|Z], clearly fail; they fail regardless of what X or Z really are, so we do not even have to generate them.

To follow the Prolog's hint, we define party determined lists in OCaml: boolean lists with a nondeterministic spine.

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

We introduce nil and cons as easy-to-use constructors of lists and a function to convert blists into ordinary OCaml lists to show them. Sample lists t3 and f2 will be used in the examples.

```
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 )
```

The append is defined as an ordinary recursive *function*, which pattern-matches on the list.

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

Here is an example of its use:

```
reify_part   None (fun () →
  list_of_blist    (append t3 f2))
⇝ [(1.,  [true; true; true; false ; false ])]
```

giving the expected result. We have defined append as a function, and can indeed run it as the concatenation function, 'forwards'.

Prolog also lets us concatenate lists that are partially or wholly unknown, represented by logic variables. For example, append([t,t,t],X,R) will enumerate all lists with [t,t,t] as the prefix, see §2. 1. If we are interested in only boolean lists, we had to complicate the Prolog code

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

and faced the problem of incomplete search: Prolog could not produce any lists that included f. In Hansei, the role of logic variable as the representation for *some* boolean list is played by a generator:

```
let rec a_blist  () : blist  =
  letlazy  (fun () →
    uniformly [|  Nil ;
                  Cons(flip  0.5,  a_blist  ()) |])
```

We need the magical function *letlazy*, which at first blush looks like the identity function. It is another primitive of Hansei, taking a thunk and returning a thunk. When we force the resulting 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, it is called "wave-function 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. Like the quantum-mechanical entanglement, letlazy is a way to share the non-deterministic state.

Passing a_blist as the second argument of append,

```
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])]
```

lets us see, within the given search bound, all boolean lists whose first three elements are **true**. Unlike the Prolog code, we are no longer stuck generating lists whose all elements are **true**.

The moment of truth is running append backwards. We have already explained the key idea of generate-and-test in §3. 1. The code in that section is trivial to adapt to partially determined lists blist; we only need the comparison function on blists:

```
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
```

Applying the generate-and-test idea to reverse the append:

```
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)
```

gives the same, expected result as in §3. 1, but with a surprise. First, the result is produced 1000 times faster. Second, the program terminates with the expected six answers even though we imposed no search bound: the first argument of reify_part is None. Although x and y in the code will generate any boolean list, thanks to laziness, the search space is effectively finite, and quite small. Thus the real speed-up due to laziness is infinite.

The letlazy operation in the definition of a_blist is crucial:

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

The operation uniformly guesses at the top constructor of the list: Nil or Cons; letlazy delays the guess, letting the program proceed until the result of the guess is truly needed. Hopefully the program rarely gets to that point because the search encountered a contradiction at some other place.

Laziness in non-deterministic computations is hence indispensable. Non-deterministic laziness however is different from the familiar facility to delay the computation and memoize its result, such as OCaml's **lazy**, Scheme's delay or Haskell's lazy evaluation. We may 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 cache the result, we should use different memo tables for different worlds, because different worlds have different choices. Ordinary lazy evaluation is implemented by mutation of the ordinary, or global, or shared memory – shared across all possible worlds. Non-deterministic laziness needs *world-local* memory [12].

4 Parsing with committed choice

Kleene star is an intrinsic operator in regular expressions and is commonly used in EBNF and other grammar formalisms. Just as common is the so-called "maximal munch" restriction on the Kleene star, forcing the longest possible match. After reminding why maximal munch is so prevalent, we describe the grave problem it poses for parsers that are meant to be run both forwards and backwards – that is, to parse a given stream according to the grammar and to generate all parseable streams, the grammar's language.

Maximal munch cuts shorter-match choices and reduces non-determinism – hence making forward runs faster. On the downside, when running the parser backwards the cut choices mean lost solutions and the (greatly) incomplete language generation. Hansei removes the downside. Parsers built with the Hansei parser combinator library support maximal munch *and* can be run effectively backwards to generate the complete language, without omissions. Surprisingly, Hansei already had the necessary features, in particular, the nested inference.

5 Maximal munch rule

The maximal munch convention is so common in parsing that it is hardly ever mentioned. For example, a programming language specification cliche defines the syntax of an identifier as a letter followed by a sequence of letters and digits, or, in the extended BNF,

```
identifier  ::= letter letter_or_digit*
```

where *, the Kleene star, denotes zero or more repetitions of letter_or_digit. In the string "var1␣+␣var2" we commonly take var1 and var2 to be identifiers. However, according to the above grammar every prefix of an identifier is also an identifier. Therefore, we should regard v, va and var as identifiers as well. To avoid such conclusions and the need to complicate the grammar, the maximal munch rule is assumed: letter_or_digit* denotes the *longest sequence* of letters and digits. Without the maximal munch, we would have to write

```
identifier  ::= letter letter_or_digit*
                    [look−ahead: not letter_or_digit ]
```

It is not only awkward, requiring the notation for look-ahead, but also much less efficient. If * means mere zero or more occurrences, letter_or_digit* on input "var1␣" will match the empty string, "a", "ar" and "ar1". Only the last match leads to the successful parse of the identifier, recognizing var1. Maximal munch cuts the irrelevant choices. It has proved so useful that it is rarely explicitly stated when describing grammars.

5.1 Maximal munch in Prolog: Reversibility lost

Maximal munch however destroys the reversible parsing, the ability to run the parser forward (as a parser or recognizer) and backward (as a language generator). We illustrate the problem in Prolog. A recognizer in Prolog is a relation between two streams (lists of characters) S and Srem such that Srem is the suffix of S. In a functional language, we would say that a recognizer recognizes the prefix in S, returning the remaining stream as Srem. Here is the recognizer for the character 'a':

```
charA([a|Srem],Srem).
```

The Kleene-star combinator (typically called many) takes as an argument a recognizer and repeats it zero or more times. Without the maximal munch, it looks as follows:

```
many0(P,S,S).
many0(P,S,Rest) :− call(P,S,Srem), many0(P,Srem,Rest).
```

where P is an arbitrary parser. Thus many0(charA,S,R) will recognize or generate the prefix of S with zero or more 'a' characters. (Recall that call is the standard Prolog predicate to call a goal indirectly: call(charA,S,R) is equivalent to the charA(S,R).) Thanks to the first clause, many0(P,S,R) always recognizes the empty string. Here is how we recognize a* in the sample input stream [a,a,b]:

```
?− many0(charA,[a,a,b],R).
 R = [a, a, b] ;
 R = [a, b] ;
 R = [b]
```

and generate the language of a*:

```
?− many0(charA,S,[]).
 S = [] ;
 S = [a] ;
 S = [a, a] ;
 S = [a, a, a] ; ...
```

To implement the maximal munch, many should call the argument parser as long as it succeeds. To tell if the parser fails or succeeds we turn to soft-cut. Recall, soft-cut P *→Q; R is equivalent to the conjunction P, Q if P succeeds at least once. Soft-cut *commits* to that choice and totally discards R in that case. R is evaluated only when P fails from the outset. Soft-cut lets us write many with maximal munch:

```
many(P,S,Rest) :−
   call (P,S,Srem) *→ many(P,Srem,Rest) ; S = Rest.
```

Now the the empty string is recognized (i.e., S = Rest) *only* if the parser P fails. Recognizing a* in the sample input

```
?− many(charA,[a,a,b],R).
 R = [b].
```

becomes quite more efficient. There is only one choice, for the longest sequence of as. However, attempting to generate the language a^*:

```
?− many(charA,S,[]).
  <loops>
```

leads to an infinite loop. The argument recognizer, charA, when asked to generate, always succeeds. Therefore, the recursion in many never terminates. When running backwards, the recognizer tries to generate the longest string of as – the infinite string. Although the empty string belongs to the language a^*, we fail to generate it.

5.2 Maximal munch in Hansei: Reversibility regained

The Hansei parser combinator library, Figure 3, supports many, which, unlike the one in Prolog, no longer forces the trade-off between efficient parsing and generation. Hansei's many obeys maximal munch *and* generates the complete language, with no omissions. Hansei lets us have it both ways. Before describing the implementation, we show a few representative examples, Figure 2.

Examples 5-7 show the argument parsers with choices, even overlapping choices as in Example 7. The combinator many (actually, many1 defined as many1 p = p <* many p) may nest. In Example 3, a∗a does not recognize "aaa" since the a∗ munches the entire stream leaving nothing for the parser of the final a. This is the expected behavior under maximal munch. Example 7 shows no parse for the same reason. Finally in the last example we generate the complete language for a^*, including the empty string.

To implement the maximal munch in Hansei we need something like soft-cut, the ability to detect a failure and proceed. Hansei has exactly the right tools: reify0 and reflect:

```
type α vc = V of α
          | C of (unit → α pV)
and α pV = (prob ∗ α vc) list

val reify0 : (unit → α) → α pV
val reflect : α pV → α
```

A parser takes a stream and returns the parsing result, the result of a semantic action, and the remainder of the stream:

```
type stream_v = Eof | Cons of char ∗ stream
and stream = unit → stream_v
type α parser = stream →α ∗ stream
```

Primitive parser

```
val let p_char : char → char parser
```

checks the current element of the stream is the given character, returning it.

Parsing combinators

```
val (<*>) : (α → β) parser → α parser → β parser
val ( <* ) : α parser → β parser → α parser
let (<|>) : αparser → α parser → α parser =
  fun p1 p2 st → uniformly [| p1;p2| ] st
```

combine parsers and their semantic actions and express the rules of the grammar: (<*>) combines parsers sequentially and (<|>) expresses the alternation. The combinator (<*) is the specializations of (<*>).

図 3 Hansei parser combinator library

The primitive reify0 converts a probabilistic computation to a lazy tree of choices αpV, whose nodes contain found solutions V x or not-yet-explored branches. The primitive reflect, the inverse of reify0, turns a tree of choices into a probabilistic program that will make those choices. The primitive reify0 is fundamental in Hansei: probabilistic inference is implemented by first reifying a program (the generative model) to the tree of choices and then exploring the tree in various ways. For instance, the full tree traversal corresponds to exact inference.

Soft-cut can also be implemented as a choice-tree traversal, first_success below, which explores the branches looking for the first V leaf. It returns the tree resulting from the exploration, which could

	Parser	Stream	Result
1	many (p_char 'a')	"aaaa"	unique
2	many (p_char 'a') <*> p_char 'b'	"b"	unique
3	many (p_char 'a') <* p_char 'a'	"aaa"	no parse
4	many (p_char 'a') <*> many (p_char 'a'))	"aaa"	unique
5	many ((p_char 'a') <|> (p_char 'b'))	"ababb"	unique
6	many ((many1 (p_char 'a')) <|> (many1 (p_char 'b')))	"aaabab"	unique
7	many ((p_char 'a' <* p_char 'a') <|> p_char 'a') <* p_char 'a'	"aaa"	no parse
8	many (p_char 'a')	random	"", "a", "aa", ...

図 2 Examples of maximal munch parsing

be empty if no V leaf was ever found. Soft-cut then is simply

```
val first_success : α pV → α pV

let soft_cut :
  (unit → α) → (α → ω) → (unit → ω) → ω =
  fun p q r →
    match first_success (reify0 p) with
    | [] → r ()
    | t → q (reflect t)
```

We write many in terms of the soft-cut as we did in Prolog:

```
let many : α parser → α list  parser = fun p →
  let rec self st =
    soft_cut
      (* check if p succeeded *)
      (fun () → p st)
      (* continue with p *)
      (fun (v, st) →
        let (vs, st) = self st in
        (v:: vs, st))
      (fun () → ([], st))   (* if p failed *)
  in self
```

The second question is avoiding losing solutions when running the parser "backwards". Unlike Prolog, Hansei parsers are functions rather than relations. They take a stream, attempt to recognize its prefix and return the rest of the stream on success. They cannot be run backwards. However, we achieve the same result – producing the set of parseable streams – by generating all streams, feeding them to the parser and returning the streams that parsed completely. Since the number of possible streams is generally infinite, we have to generate them lazily, on demand. To ensure completeness – to avoid losing any solutions – the parsers should have the property

```
many p (s1 ⊕ s2) = many p s1 ⊕ many p s2
```

where ⊕ stands for non-deterministic choice. Surprisingly, many p already satisfies it. The trick is laziness in the stream and Hansei's support of nested inference. The primitive reify0 may appear in probabilistic programs – in other words, a probabilistic model may itself perform inference, over an inner model. In order for this to work correctly, we had to ensure that

```
let x = letlazy (s1 ⊕ s2) in
        reify0 (fun () → model x)
≡
let x = letlazy s1 in reify0 (fun () → model x) ⊕
let x = letlazy s2 in reify0 (fun () → model x)
```

where x is demanded in model. That is, reify0

should reify only the choices made by the inner program, and let the outer choices take effect. The stream generator has to be lazy, so it has the form letlazy (s1 ⊕ s2). Comparing the nested inference property with the code for many reveals that the key property of many p (s1 ⊕ s2) is satisfied without us needing to do anything. Our many has exactly the right semantics.

6 Conclusions

Classical logic programming all too often forces us to choose between efficiency and expressiveness, on one hand, and completeness on the other hand. Negation and committed choice make logic programs easier to write and, in some modes, faster to run. Alas, some other modes (informally, running 'backwards') become unusable or impossible. Kleene star is a good example of the trade-off: maximal munch simplifies the grammar and makes parsing efficient, but destroys the ability to generate grammar's language. Functional logic programming systems can remove the trade-off. Properly implemented encapsulated search (nested inference, in Hansei) lets us distinguish the choices of the parser from the choices of the stream and cut only the former. Perhaps surprisingly this distinction just falls out of the need for non-deterministic stream to be lazy. Thus Kleene star with maximal munch lets us parse and generate the complete language. In Hansei, we can have it both ways.

We have gone back to Herbrand: we build the Herbrand universe (the set of all ground terms) and explore it to find a model of a program. We build the universe by modeling 'logic variables' as generators for their domains. Since the Herbrand universe for most logic programs is infinite, non-strict evaluation is the necessity. Furthermore, since logic variables may occur several times, we must be able to correlate the generators. Finally, we need a systematic way of exploring the search space, without getting stuck in one infinite sub-region. Hansei, among other similar systems, satisfies all these requirements.

Logic variables and unification have been introduced by Robinson as a way to 'lift' ground resolution proofs of Herbrand, to avoid generating the vast number of ground terms [13]. Logic variables effectively delay the generation of ground terms to the last possible moment and to the least extent. Doing computations only as far as needed is also the goal of lazy evaluation. It appears that lazy evaluation can make up for logic variables, rendering Herbrand's original approach practical. It remains a fascinating task to be able to systematically derive a unification procedure.

参 考 文 献

[1] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer, 5 edition, 2003.

[2] Peter Van Roy. 1983-1993: The wonder years of sequential prolog implementation. Technical Report 36, Paris DEC Research Laboratory, December 1993.

[3] Fernando C. N. Pereira and Stuart N. Shieber. *Prolog and Natural-Language Analysis.* Center for the Study of Language and Information, Stanford, California, 1987.

[4] Patrick Blackburn, Johan Bos, and Kristina Stiegnitz. *Learn Prolog Now!* College Publications, London, 2006.

[5] T. Moto-oka, editor. *Fifth Generation Computer Systems.* North-Holland, New York, 1982.

[6] Committee for Development and Promotion of Basic Computer Technology. Fifth generation computer systems project final evaluation report. *ICOT Journal*, No. 40, pp. 3–24, 1994.

[7] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM*

Journal of Research and Development, Vol. 3, pp. 114–125, 1959.

[8] Lee Naish. Pruning in logic programming. Technical Report 95/16, Department of Computer Science, University of Melbourne, Melbourne, Australia, June 1995.

[9] Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In Walid Mohamed Taha, editor, *Proceedings of the Working Conference on Domain-Specific Languages*, No. 5658 in LNCS, pp. 360–384. Springer, 15–17 July 2009.

[10] Oleg Kiselyov and Chung-chieh Shan. Monolingual probabilistic programming using generalized coroutines. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, pp. 285–292, Corvallis, Oregon, 19–21 June 2009. AUAI Press.

[11] Michael Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, No. 4670 in LNCS, pp. 45–75. Springer, 2007.

[12] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy nondeterministic programming. *Journal of Functional Programming*, Vol. 21, No. 4–5, pp. 413–465, 2011.

[13] John Alan Robinson. Computational logic: Memories of the past and challenges for the future. In *Proceedings of the First International Conference on Computational Logic*, No. 1861 in LNAI, pp. 1–24, 2000.