Guess Lazily!

making a program guess, and guess well

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

Strange Loop 2012
St Louis, MO September 25, 2012

Outline

Guessing is a part of life and science. We form a hypothesis, work out the consequences and compare with observations. Lots of problems are formulated by first assuming that the solution exists and then describing the properties it should have. Planning, scheduling, diagnostic, learning problems and Sudoku all follow this pattern. Guessing is good not only for describing these problems but also for solving them. We make a guess – often a series of guesses – to build, for example, a schedule, and check if it satisfies resource, timeliness and other constraints. Often, we guess again.

How do we write "guess the value of this variable" in code? How do we code "guess again"? How to put in prior knowledge favoring some guesses? The talk first will answer these questions.

Naive guessing however is hopeless even for toy problems. We often have to make lots of guesses before we build a candidate solution to check against the constraints. Only a tiny or even infinitesimal proportion of these guesses yield a successful candidate. How to make good guesses? That is very hard to know: Most physical, biological, sociological, etc. models are set up to compute consequences of the causes rather than the other way around. It helps to reformulate the question: how to avoid too many bad guesses? The talk will describe and illustrate a general principle, found in any serious logic, non-deterministic or probabilistic programming system.

The techniques explained in the talk are not tied to any language and can be used even in C. However, functional, especially typed functional languages have a serious advantage, as we shall see. No prior knowledge of logic or non-deterministic programming is required. The ability to read introductory OCaml or Haskell code will be helpful. The participants will learn how to guess in their favorite language, and what it takes to succeed doing so. They will see laziness, unification and constraint propagation in the same light.

Introduction

How to guess

- Specifying problems by way of a guess
- Solving problems by guessing

Non-deterministic programming in many languages

How to guess well

The two main topics for the talk are how to guess and how to guess well. Very many practical problems are posed in a form that involve a guess. For example, we assume that a schedule exists and then enumerate its desired properties. Many non-practical problems are also posed that way: for instance, the N-queen problem says: *guess* the positions of N queens *such that* they don't kill each other. The whole class of NP problems are formulated that way. Recall that N in NP stands for non-determinism. We guess the solution, and then verify in polynomial time, that it is correct. But non-determinism is good not only for asking questions. It is also a good way of answering them. So, the first part of the talk is about non-deterministic programming in many languages.

Introduction

How to guess

- Specifying problems by way of a guess
- Solving problems by guessing

Non-deterministic programming in many languages

How to guess well

- Fail early, fail often
- Laziness

Unification and other constraint programming

As you might've *guessed*, naive guessing hardly ever successful. A very simple principle helps, and I give it away now: laziness! But not that one! Hopefully by the end you'll see what I mean and how it all relates to unification and other constraint functional-logic programming.

Outline

We start with a simple scheduling problem.

Puzzle

"U2" has a concert that starts in 17 minutes and they must all cross a bridge to get there. They stand on the same side of the bridge. It is night. There is one flashlight. A maximum of two people can cross at one time, and they must have the flashlight with them. The flashlight must be walked back and forth. A pair walk together at the rate of the slower man's pace:

Bono	1 minute to cross
Edge	2 minutes to cross
Adam	5 minutes to cross
Larry	10 minutes to cross

For example: if Bono and Larry walk across first, 10 minutes have elapsed when they get to the other side of the bridge. If Larry then returns with the flashlight, a total of 20 minutes have passed and you have failed the mission.

Allegedly, this is a question for potential Microsoft employees. An answer is expected within 5 minutes.

It is indeed a typical scheduling problem: find a sequence of decisions – who should walk in what sequence – subject to a set of constraints, optimizing some utility. We assume that there is a schedule and describe its properties.

There are two answers, neither of which are trick answers. Allegedly, this is one of the questions for potential Microsoft employees. Some people really get caught up trying to solve this problem. Reportedly, one guy solved it by writing a C program, although that took him 37 minutes to develop (compiled and ran on the 1st try though). Another guy solved it in three minutes. A group of 50, at Motorola, couldn't figure it out at all.

Simple library for non-determinism

val choose : α **list** $\rightarrow \alpha$

let fail () = choose []

A clear and elegant way of solving the puzzles like ours is non-determinism. For concreteness, I will use OCaml in this talk. I could just as well use Scala, for example. I could've even chosen C – and once I did. Anyway, OCaml isn't special in that it hasn't been designed for non-deterministic programming.

Let's assume that somehow we have this function choose that chooses an element from a list. Choosing from the empty list fails the computation.

Solving the puzzle

```
type u2 = Bono | Edge | Adam | Larry
type side = u2 list

let rec loop trace forward time_left = function
  | ([], _) when forward →
      print_trace (List.rev trace)
  | (_, []) when not forward → ...
  | (side_from, side_to) →
      let party = select_party side_from in
      let elapsed = elapsed_time party in
      let _ = if elapsed > time_left then fail () in
      let side_from' = without party side_from in
      let side_to' = side_to @ party in
      loop ((party, forward)::trace) (not forward)
           (time_left - elapsed) (side_to', side_from')
```

This code represents the specification of the problem, in the most straightforward way. We keep a list of people on both sides of the bridge, and let them walk with the flashlight back and forth. We finish when everyone made it or when the time is up. This is the totally standard OCaml code. I'm sure everyone in the audience can write such code in their sleep. Perhaps only one function would give a pause.

Selecting a party

```
let select_party side =
  let p1 = choose side in
  let p2 = choose (List.filter (fun x → x ≥ p1) side) in
  if p1 = p2 then [p1] else [p1;p2]
```

But even the selection function is most straightforward, if we could non-deterministically select an element from the list. And our simple library provides exactly that function. The code reflects that the order of people within a pair is not important.

Implementing non-determinism

```
let rec choose = function
    | []      → exit 666
    | [x]     → x
    | (h:: t) →
        let pid = fork () in
        if pid = 0 then h
        else wait (); choose t

let run m = match fork () with
    | 0 → m (); printf "Solution␣found"; exit 0
    | _ → try while true do waitpid [] 0 done
          with ...
```

One way to implement non-determinism is just to run all the choices, perhaps in parallel, and hope one of them eventually succeeds. At the point of making a choice, we split the computation into several parts. Each split-off computations proceed with one of the choices. Everyone here knows how to split the computations: use fork.

It indeed works. It is interesting to watch, using top, how processes are launched and how they die, how their number increases and drops.

Implementing non-determinism

```
let rec choose = function
    | []      → exit 666
    | [x]     → x
    | (h:: t) →
        let pid = fork () in
        if pid = 0 then h
        else wait (); choose t

let run m = match fork () with
    | 0 → m (); printf "Solution␣found"; exit 0
    | _ → try while true do waitpid [] 0 done
          with ...
```

I'd like to point out the `fork` in `run`: we split the computation into a process that does all the work, and the supervisor. As in real life, the supervisor immediately goes to sleep. It wakes up when all the workers are finished, to report the achieved result or an exception.

As we watch: ocamlrun is a byte-code OCaml interpreter. Watching processes coming and going really drives home how much non-determinism is involved. This problem would work great on a Hadoop cluster or in a cloud. I think you see that my mentioning of C wasn't a joke: you can do non-deterministic programming on any language that has a POSIX FFI, to fork. Almost any language will do.

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
...

Well, I hope it was entertaining. You might be thinking that fork () was a bit baroque. Don't be too quick to ridicule: there is an important lesson here, as we shall see soon. But before we do, we should arrange for a faster choice: instead of a heavy-weight Unix fork(), we need a *green fork*. Luckily, OCaml has exactly the right thing, which underlies a library for probabilistic programming, called Hansei. We will use this library, but hush the probabilities.

The library has the basic functions dist, which is basically choose, but with probabilities, 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, of the probability table.

Outline

Parsing

```
type stream_v = Eof | Cons of char * stream
and  stream   = unit -> stream_v

type α parser = stream -> α * stream
                (* non-deterministically *)
```

To recap the first point of the talk and transition to the second one, let's consider another, more realistic example: parsing. It, too, benefits from non-determinism. The non-deterministic choice is prominent in BNF grammars. To illustrate, let me show a simple parsing combinator library: something like Parsec, which is available for many languages.

As usual, parsers parse a stream of characters; the characters do not need to be present all in memory, but can be read on demand. That's why stream is a thunk. A parser takes a stream and returns the parsing result (the result of a semantic action) and the remainder of the stream.

The type alas does not make it explicit that parsers are non-deterministic.

Primitive Parsers

```
val empty : unit parser

let p_sat : (char → bool) → char parser = fun pred st →
  match st () with
  | Cons (c, st) when pred c → (c, st)
  | _                        → fail ()

let p_char : char → char parser =
  fun c → p_sat (fun x → x = c)
```

And parsers are generally non-deterministic. First of all, they may fail. Here are the primitive parsers in our library (like in many others). The parser `empty` parses the empty string. The parser `p_sat` checks if the current character satisfies a given predicate. A parser for a character `p_char` is written in terms of it. If the current character does not satisfy the predicate, we fail.

Parser Combinators

let $(<|>)$: αparser $\rightarrow$ α parser $\rightarrow$ α parser = **fun** p1 p2 st $\rightarrow$
uniformly [|p1;p2|] st

let alt : α parser **array** $\rightarrow$ α parser = **fun** pa st $\rightarrow$
uniformly pa st

val $(<*>)$: $(\alpha \rightarrow \beta)$ parser $\rightarrow$ α parser $\rightarrow$ β parser
val $(<\$>)$: $(\alpha \rightarrow \beta) \rightarrow$ α parser $\rightarrow$ β parser
val $(*>)$: α parser $\rightarrow$ β parser $\rightarrow$ β parser
let $(<*)$: α parser $\rightarrow$ β parser $\rightarrow$ α parser

val p_fix : $(\alpha$ parser $\rightarrow \alpha$ parser$)$ $\rightarrow$ α parser
val many : α parser $\rightarrow$ α **list** parser

Parser combinators combine parsers and their semantic actions. For example, `<*>` hooks up the parser for a prefix with the parser for the rest of the stream, and combines the corresponding semantic actions. You can guess what the other parsers at the bottom do from their types. More interesting is `<|>`, for parsing alternatives. It is indeed implemented as a non-deterministic choice of a parser for the rest of the stream.

Example: recognizing palindromes

```
let pali = p_fix (fun pali →
  alt [| empty;
         (fun _ → ()) <$> p_char 'a';
         (fun _ → ()) <$> p_char 'b';
         p_char 'a' *> pali <* p_char 'a';
         p_char 'b' *> pali <* p_char 'b' |]
    )

run_fwd pali "ab";;
run_fwd pali "abaaba";;
```

Here is the first example: recognizing palindromes. Since we build a recognizer, the semantic actions do nothing, returning unit. The grammar reads pretty much like BNF, doesn't it? A palindrome over the two-character alphabet is *either* the empty string, a single character, or a palindrome flanked on both sides with the same character. We can run a few examples.

Show the definition of run_fwd and show the two examples.

Example: generating palindromes

```
run_bwd (Some 10) pali (fun () → stream_over [|'a';'b'|])

run_bwd None pali (fun () →
  let st = stream_over [|'a';'b'|] in
   stream_len st 5 ; st )
```

It is still OCaml – not Prolog, not Kanren, not Monads

How come the second example terminates?

I guess you might've seen where all this was going. We can run the *same* parser not only forwards but also backwards, not only recognize palindromes but also to generate them.
(Show the examples live). We do that by making the input stream to be non-deterministic as well. The function stream_over generates an *all* streams over a given alphabet, including non-terminated ones.
(Show what the generator does.) We cannot use exact_reify since the search tree is infinite. We have to limit the depth of the search, say, to 5 levels.
Let's parse all possible streams of a's and b's, and see what we get. We must limit our search, because it won't ever end. There is an infinite number of palindromes.
The next example finds all 5-letter palindromes. We add the predicate stream_len st 5 that fails if the stream does not have 5 elements. Now we can use the exhaustive search. As a bonus, all palindromes are generated equally likely, with no bias.

Example: parsing and generating arithmetic expressions

```
run_fwd expression "10"
⇝ [(0.125, Ptypes.V 10)]

run_fwd expression "(10+ 5*2)/4"
⇝ [(1.52587890625e−05, Ptypes.V 5)]

(* and running backwards *)
```

Related: generating random C code to test C compilers

Before we answer the termination question, let's look at another example: parsing arithmetic expression, with a semantic action to compute its value. The grammar does look like that one printed in probably every book on compiler construction. We can run the parser forward, and backwards.

I have tried not to talk about probabilities, but you have probably guessed what the first number means. Incidentally, probabilities could be used to set prior knowledge. The inverse of the probability is the estimate of the number of possible worlds that had to be examined.

Running backwards in detail

```
exact_reify  pali  (fun () →
  let  st  = stream_over [|'a';'b'|] in
   stream_len  st  5;
  let  (v, st′ ) = pali  st  in
  if  st′  ()  ≠ Eof then fail  ();
  (v, string_of_stream  st ))
```

How could it possibly work and terminate?

Let's come back to the code that finds all palindromes of length 5. The code seems straightforward: choose a sample sequence of a's and b's. If the length isn't 5 or if it doesn't completely parse as a palindrome, fail. If we made it to the end, report the sequence of characters as a text string.

But why does the code terminate? If stream_over generates *all* sequences of a's and b's, and stream_len st 5 rejects them (except the ones of length 5), the search will get stuck when choosing longer and longer streams and failing them all. But it doesn't. Why?

Uneager stream

```
type stream_v = Eof | Cons of char * stream
and  stream   = unit -> stream_v

let  rec stream_len  st  n = match (st ()  , n) with
   |  (Eof,0)                          -> ()
   |  (Cons (_, t), n) when n > 0 ->stream_len  t  (n-1)
   |  _                                -> fail   ()
```

stream_len st 5 forces no more than 6 thunks

Less haste, *infinitely* more speed

Recall the type of the stream. Remember I said that all characters don't have to be present in memory, they can be read on demand. They also can be chosen on demand. The entire tail of the stream can be chosen on demand.

The code for stream_len shows this demand, when forcing a thunk. It is easy to see that stream_len st 5 forces no more than 6 thunks. If the 6th thunk is a Cons, we fail and never examine any part of the stream after that. The rest of the stream is never demanded, and is never chosen.

We were not eager to generate the sequence before examining it, delaying the choices until we need to look at it. The result is quite a dramatic improvement: infinite improvement. The search problem became finite.

Running backwards in detail

```
exact_reify  pali  (fun () →
  let  st  = stream_over [|'a';'b'|] in
  stream_len  st  5;
  let  (v, st′) = pali  st  in
  if  st′  ()  ≠ Eof then fail  ();
  (v, string_of_stream  st ))
```

How could it possibly work and terminate?

As if st is the same choice of a stream, not just the same *procedure* of choosing a stream

But there is another problem: for example, string_of_stream does not have any size limitation; it tries to convert the entire stream into a string. Ditto for the parser. Furthermore, when a parser forces a choice, the result can generally differ from that of string_of_stream forcing the same thunk. Flipping the same coin generally gives different result. But the code is written as if sf were the same choice of a stream, not just the same procedure of choosing a stream. Is there magic? In a sense, yes.

Running backwards in detail

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

let stream_over : char array → stream = fun ca →
  let rec loop () =
    if flip 0.5 then Cons (uniformly ca, letlazy loop)
                else Eof
  in letlazy loop
```

- call-time choice
- wave-function collapse

Let me disclose the code the chooses a stream over a given alphabet. You see the magical function *letlazy*, which at first blush looks like an identity function. 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.

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

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

Type inference: forwards and backwards

Simply typed λ-calculus with integer literals

```
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
```

Obtain a type for a term, or terms for a type, or all well-typed terms

The type-inference code fits on a single slide. The code looks quite like the familiar typing rules. The syntax of the language – the AST – should be clear from the pattern-matching. We can either obtain the type for a term, or generate all terms for a given type or all well-typed terms.

Outline

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.

Buzzwords

- non-deterministic choice
- fork ()
- Schrödinger cat
- wavefunction collapse
- first-class storage
- call-time choice
- thread-local memory
- *logic variable*
- unification
- constraint propagation
- generate-and-test
- test-and-generate

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...