15–150: Principles of Functional Programming

Games

Michael Erdmann*

Spring 2020

In the this lecture and the next, we will implement a program that plays any two-player, deterministic, perfect-information, finitely branching, zero-sum game. The *minimax* algorithm for playing such games is a standard example in AI, and we will implement it in SML. In fact, that algorithm is the basis for many ideas in automated planning and machine learning. Generalizations of the algorithm described here apply in settings with imperfect information and uncertain outcomes, such as planning the motions and actions of a robot.

The code development in the next two lectures provides a nice use of modules, and in particular an elegant application of functors for code re-use. The code also illustrates some features of SML that we haven't talked about yet: views; mutual recursion; input/output; sharing constraints.

1 Overview

Games

What is a two-player, deterministic, perfect-information, finitely branching, zero-sum game?

- "Two-player" means the game is played by two players who alternate taking turns.
- "Deterministic" means that each move has a well-defined outcome; there is no randomness (no luck, no rolling the dice).
- "Perfect-information" means that, at any given moment, both players know the complete state of the game; there is no hidden information.
- "Finitely branching" means that there are only a finite number of allowed moves at each stage of the game.
- "Zero-sum" means that if I win, you lose, and vice versa—what's good for me is bad for you. Draws are allowed.

Examples include tic-tac-toe, Nim, chess, checkers, Connect 4, Mancala, and Gomoku. By way of contrast, poker is not a perfect-information game, because neither player knows the cards held by the other and because there is randomness in the draws.

*Adapted from documents by Stephen Brookes and Dan Licata.

Nim

Let's illustrate with Nim: Let's start with 15 pieces. You make the first move: you pick up 1, 2, or 3 pieces. Then it's my turn. I have to pick up 1, 2, or 3 pieces. Then it is your turn again. And so forth. Whoever picks up the last piece loses.

Here is an example:

```
We start with 15 pieces
You pick up 3 (12 pieces left)
I pick up 3 (9 left)
You pick up 2 (7 left)
I pick up 2 (5 left)
You pick up 1 (4 left)
I pick up 3 (1 left)
You pick up 1 (0 left)
I win!
```

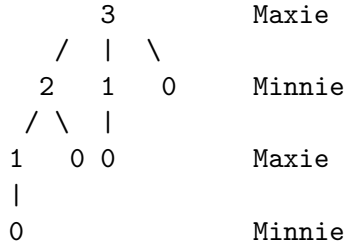
In fact, Nim with 15 pieces has a winning strategy for whoever goes first: To understand this, let's suppose we are down to 5 pieces. If it is your turn, and there are 5 pieces left, then you lose: go ahead and try it. If you take 1, I take 3, and there is 1 left. If you take 2, I take 2. If you take 3, I take 1. No matter what you do, there is 1 left on your next turn. Similarly, if there are 9 pieces left and it is your turn, then I can ensure that on your subsequent turn there will be 5 pieces left, similarly from 13 to 9, etc. What is the pattern? If it is your turn, and the number of pieces (`mod` 4) is 1, then I have a winning strategy. So I choose my move to always leave the number of pieces congruent to 1 (`mod` 4).

Nim is special, in that I can do a quick calculation that tells me who will win. For chess, one can not tell (in constant time) just by looking at the board who will win. So, if you were writing a program to play it, what would you do? Use your computational resources to explore possible future states!

Game Trees

To explore possible futures in a game, one imagines a *game tree*. The tree's nodes represent game states and its edges represent game moves. Each layer in the tree is labeled with the name of the player whose turn it is in all the states at that level. In what fallows, we will call the players `Maxie` and `Minnie` to emphasize the role of maximization and minimization in the upcoming minimax algorithm. The SML code implementing this algorithm won't actually build trees like this; instead, it will explore possible futures recursively, in a way that mimics the tree structure.

Again, let's illustrate with Nim, starting with 3 pieces to keep the tree small enough to see well.



The nodes are Nim states (number of pieces), and each downward edge represents the effect of making a Nim move at a state (i.e., removing 1, 2, or 3 pieces).

We want to assign to each state a value which tells us who (eventually) wins from that state. We use the value 1 to indicate a win for `Maxie`, and -1 to indicate a win for `Minnie`.

First, we label the leaves. Leaf nodes for Nim have 0 pieces. If there are 0 pieces left, and it's my turn, then you took the last one, so I won. We'll put `?` in nodes to which we have not yet assigned a value.

```
                 (3,?)            Maxie
              /    |     \
          (2,?)   (1,?)   (0,-1)   Minnie
          /  \      |
(1,?)  (0,1)  (0,1)              Maxie
  |
(0,-1)                           Minnie
```

Next, we propagate these values up the tree. If it's `Maxie`'s turn at a node, then the value is the maximum value of the children, assuming all children have been assigned values. If it's `Minnie's turn`, we take the minimum.

First level: (we propagate from the deepest leaf to its parent node)

```
                 (3,?)            Maxie
              /    |     \
          (2,?)   (1,?)   (0,-1)   Minnie
          /  \      |
(1,-1) (0,1)  (0,1)              Maxie
  |
(0,-1)                           Minnie
```

Next level:

```
                 (3,?)            Maxie
              /    |     \
         (2,-1)   (1,1)   (0,-1)   Minnie
          /  \      |
(1,-1) (0,1)  (0,1)              Maxie
  |
(0,-1)                           Minnie
```

After this step we see in the leftmost child tree that if there are 2 pieces left, Minnie should take 1, rather than 2: taking 1 piece leads to state $(1, -1)$ whose value -1 says that Minnie wins, whereas taking 2 pieces leads to a state that is a win for Maxie.

Finally, top level:

```
                 (3,1)            Maxie
              /    |     \
         (2,-1)   (1,1)   (0,-1)   Minnie
          /  \      |
(1,-1) (0,1)  (0,1)              Maxie
  |
(0,-1)                           Minnie
```

The information at the root now says that when starting with 3 pieces Maxie should take 2 pieces in order to win, leaving 1, rather than taking 1 piece or 3 pieces.

Exercise: draw and label the Nim game tree starting from 5 pieces.

What the *minimax algorithm* (as illustrated above) computes is the value of a game state, assuming both players will play optimally. It does not account for imponderables like "if I do this, the chess board will look more confusing, so I think you're likely to make a mistake".

In a game with a bigger search space, we cannot draw out the whole tree, or expand out the recursive calls forever! Instead, we will write a heuristic that looks at states and approximate their value. For Nim, there is a perfect heuristic: *is the number of pieces congruent to 1 modulo 4*? For chess, a useful heuristic would take account of which pieces are left, where they are positioned, etc. This is where the smarts in playing a particular game come in. Then, the overall algorithm for selecting a move is to (a) explore the game tree up to a certain depth, (b) use the heuristic to approximate node values at that depth, and (c) propagate those values back up to the root of the tree as we just did using the minimax algorithm.

2 Game Architecture

The process of minimax game tree searching is independent of the particular game. Moreover, the process of putting together a run of a game, given two players, is independent of the game and the players. We represent this by defining some signatures:

```
signature GAME
signature PLAYER
```

We can define various `GAME`s, like Chess, Connect 4, Mancala, Othello. We can define other types of "players", that is, search algorithms, beyond the minimax described above. For instance, in the next lecture we will see *alpha-beta*. This is a search algorithm that prunes parts of the search space that are known to be suboptimal. We can also set up our interface to include a human player. Each such player will have a generic component that can play any game, and a heuristic component specific to the particular game being played. And we can also define a generic referee that puts two players together and runs a game. This is an example of *modular program design*, where we will use functors to achieve good code reuse. And it is an application-specific illustration of modules.

3 Games

Let's start with the signature for a game. As usual, we assume that we have a structure `Seq:SEQUENCE` implementing sequences.

```
signature GAME =
sig
    datatype player = Minnie | Maxie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

    type state  (* abstract, representing states of the game *)
    type move   (* abstract, representing moves of the game  *)
```

```
    val start : state

    (* REQUIRES:  m is in moves(s)                *)
    (* ENSURES:   make_move(s,m) returns a value.  *)
    val make_move : state * move -> state

    (* The next three functions are "views" of the abstract types state and move. *)

    (* REQUIRES: status(s) == In_play                                        *)
    (* ENSURES:  moves(s) returns a nonempty sequence of moves legal at s. *)
    val moves : state -> move Seq.seq

    val status : state -> status
    val player : state -> player   (* says whose turn it is to make a move *)

    datatype est = Definitely of outcome | Guess of int

    (* REQUIRES:  status(s) == In_play          *)
    (* ENSURES:   estimate(s) returns a value.  *)
    val estimate : state -> est

    [plus helper functions useful for input/output, omitted ]

end
```

In words, a game structure ascribing to the `GAME` signature must provide:

- A datatype of two players, with standard names `Maxie` and `Minnie`. This is a feature of signatures that we haven't exploited previously: you can put a datatype definition in a signature, in which case clients have access to the constructors. These constructors may be used to create values and for pattern-matching. Also in order to implement this signature you *must* declare exactly the same datatype.
- Two datatypes `status` and `outcome` whose values tell you whether the game is over, and, if it is, what the outcome was. These types are again datatypes with standard (not game-specific) constructors (`Over` and `In_play` for status; `Winner` and `Draw` for outcomes).
- An abstract type `state` whose values represent game states, including the board, whose turn it is, etc.
- An abstract type `move` representing an action a player can take.
- A start state `start`.
- A transition function `make_move` that applies a move to a state, returning the resulting state. The move must be an action allowed at the state.
- A function `moves` that computes the sequence of allowed actions at a given state.

- A function `status` that tells us if a game state represents a completed game, and if so, what the outcome was.
- A function `player` that tells us whose turn it is in a given state.

  These last three functions are called *views*, because they allow us to see information about values of the abstract types `state` and `move`.
- A game comes with a type `est` and a function `estimate`, that approximates the value of a state. An estimation produces either `Definitely` an outcome, or a `Guess` of an integer; a positive guess indicates "better for Maxie", whereas a negative guess indicates "better for Minnie". The magnitude of a guess indicates how favorable the estimate looks.

  We regard estimate values as an ordered type, with a greater-than ordering based on the following:

  ```
  Definitely (Winner Maxie)    >
  Guess(some positive number)  >
  Guess(0) and Definitely Draw >
  Guess(some negative number)  >
  Definitely (Winner Minnie)
  ```

  Comment: We have placed the estimator inside `GAME` for simplicity in these introductory lectures. More generally, one would want to place it in a separate signature/structure.
- Finally, a game comes with some parsing and printing functions, omitted here.

Multiple abstract types at once: Note that this signature defines two abstract types, for states and moves, at once. The implementation must be privy to both at once, and clients don't need to know either. This is perfectly natural in SML, but difficult in some languages.

4 Nim

Here is an implementation of Nim, eliding the parsing and printing code. Since we already saw that there is a simple way to figure out how to win at Nim, we build into this implementation a "genius" estimator that never needs to guess!

The code makes use of SML's predefined exception `Fail` that can be raised with a string argument, useful for returning error messages.

```
structure Nim : GAME =
struct
    datatype player = Maxie | Minnie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

    datatype state = State of int * player
         (* The integer component is the number of pieces left.  *)
         (* The player component is who should take pieces next. *)
```

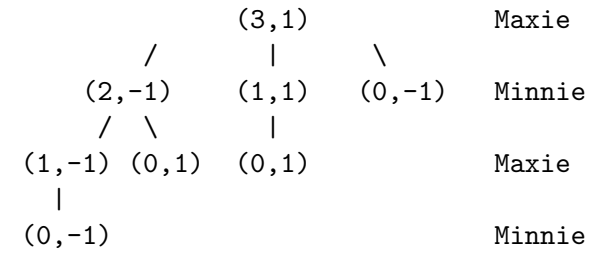
```
    datatype move = Move of int
         (* The integer is how many pieces to pick up.         *)

    val start = State (15, Maxie)
         (* Initial state for Nim has 15 pieces, Maxie goes first. *)


    fun flip Maxie = Minnie
      | flip Minnie = Maxie
         (* flip : player -> player *)
         (* Switches the player from Maxie to Minnie or vice versa. *)

    fun make_move (State (n, p), Move k) =
        if (n >= k) then State (n - k, flip p)
                    else raise Fail "tried to make an illegal move"


    fun moves (State (n , _)) = Seq.tabulate (fn k => Move(k+1)) (Int.min(n,3))

    fun status (State (0, p)) = Over(Winner p)
      | status _ = In_play
        (* Nim is over when no pieces are left.                          *)
        (* When game ends, whoever would have moved next is the winner. *)

    fun player (State (_, p)) = p


    datatype est = Definitely of outcome | Guess of int

    fun estimate (State (n, p)) =
        if n mod 4 = 1 then Definitely (Winner (flip p))
                       else Definitely (Winner p)
       (* If there are n pieces left, with n=1 (mod 4), then the player
          whose turn it is must lose, assuming optimal play by opponent.
          Otherwise, that player can win.                                 *)

    [parsing and printing, omitted]
end
```

- To satisfy a datatype declaration in a signature, you put the same datatype declaration in the structure. Pretty boring, but necessary.
- A state is represented by a pair. The first component of the pair is an integer – the number of pieces left. The second component of the pair is the player whose turn it is to remove pieces. A move is represented by an integer (which must be 1, 2, or 3), describing the number of pieces to remove.
- The types `state` and `move` are defined to be datatypes that aren't exported, so they are

abstract. This way, no one can accidentally make an invalid state (e.g., "there are `~17` pieces left").

- The start state is 15 pieces and Maxie's turn.
- For `make_move`: to apply a move, we just subtract (the move is assumed to be valid).
- For `moves`: if there are fewer than three pieces, one may only take up to the number left; otherwise one may take 1, 2, or 3.
- For `status`: if there are no pieces left, the game is over, and the player whose turn it is wins, because whoever took the last piece loses.
- For `player`: this reports the player whose turn it is. If we forget to put the player in the state, then we would not be able to implement this function. Moral: the operations to be provided place demands on the implementation of abstract types.
- The estimator just calculates *modulo* 4, and says who is definitely going to win. For Nim, the estimator *never* makes a `Guess`! In more typical games, where there may not be an easily discoverable winning strategy.

To see how the SML structure `Nim` corresponds to our earlier discussion of Nim game trees, recall the game tree we drew then:



Using the functions defined in the Nim structure, we have the following facts, each of which has a pictorial echo in this tree drawing. (We use the angle-bracket notation for sequences.)

- `moves(State(3, Maxie)) = <Move 1, Move 2, Move 3>`
- `moves(State(0, Minnie)) = <>`
- `make_move(State(3, Maxie), Move 2) = State(1, Minnie)`
- `status(State(0, Minnie)) = Over(Winner Minnie)`
- `estimate(State(3, Maxie)) = Definitely(Winner Maxie)`
- `estimate(State(2, Minnie)) = Definitely(Winner Minnie)`
- `estimate(State(1, Maxie)) = Definitely(Winner Minnie)`

In the next lecture we will implement the minimax algorithm using recursive functions, rather than with a perfect estimator. It will turn out that our implementation calculates the same values for states as the ones appearing in the tree drawing above.

5 Dumb Nim

Just for contrast, and for later when we implement bounded minimax players, the next page shows a "bad" implementation of the Nim game (with the same implementation of the types and game operations) in which the estimator is almost useless. The code is exactly the same as the structure called `Nim` above, except for the `estimate` function.

```
structure DumbNim : GAME =
struct
    datatype player = Maxie | Minnie
    datatype outcome = Winner of player | Draw
    datatype status = Over of outcome | In_play

    datatype state = State of int * player
         (* The integer component is the number of pieces left.  *)
         (* The player component is who should take pieces next. *)

    datatype move = Move of int
         (* The integer is how many pieces to pick up.         *)

    val start = State (15, Maxie)
         (* Initial state for Nim has 15 pieces, Maxie goes first. *)


    fun flip Maxie = Minnie
      | flip Minnie = Maxie
         (* flip : player -> player *)
         (* Switches the player from Maxie to Minnie or vice versa. *)

    fun make_move (State (n, p), Move k) =
        if (n >= k) then State (n - k, flip p)
                    else raise Fail "tried to make an illegal move"


    fun moves (State (n , _)) = Seq.tabulate (fn k => Move(k+1)) (Int.min(n,3))

    fun status (State (0, p)) = Over(Winner p)
      | status _ = In_play
        (* Nim is over when no pieces are left.                         *)
        (* When game ends, whoever would have moved next is the winner. *)

    fun player (State (_, p)) = p


    datatype est = Definitely of outcome | Guess of int

    (* This estimator is intentionally fairly dumb. *)
    fun estimate (State (1, p)) = Definitely (Winner (flip p))
      | estimate _ = Guess 0
       (* If there is exactly 1 piece  left, then the player
          whose turn it is must lose.  Otherwise, guess a draw. *)

    [parsing and printing, omitted]
end
```

6 Views

The functions `player` and `status` are examples of *views*: functions that map an abstract type (in this case `state`) into values of a datatype revealed by the signature, so that one may see it and use it. This kind of pattern-matching on values derived from abstract types is very useful, so let's look at some other instances of it.

As we have discussed many times, list operations have bad parallel complexity, but the corresponding sequence operations are much better. However, sometimes one may want to write a sequential algorithm (e.g., because the inputs aren't very big, or because no good parallel algorithms are known for the problem). As we have discussed the sequence interface so far, it is difficult to decompose a sequence as "either empty, or a cons with a head and a tail." To implement such a construct using the sequence operations we discussed previously, you would have to write code that would surely lose style points, such as:

```
case Seq.length s of
     0 =>
   | _ => ... (Seq.nth s 0) ... (Seq.tabulate (fn i => Seq.nth s (i+1)) ...) ...
```

It would be more desirable to use patterns to bind variables to pieces of `s`, as in

```
case s of
   Nil => ...
 | Cons(x,s') => ...
```

but we cannot pattern-match on `s` since its value belongs to an abstract type.

We can address this problem using a *view*. This means that we put an appropriate datatype in the signature, along with functions converting sequences to and from this datatype. This allows us to pattern-match on an abstract type, while keeping the actual representation abstract. (By analogy, we put datatypes `player`, `outcome`, `status` in the signature `GAME`, as well as the functions `player` and `status`. Consequently, a client can pattern-match on some components of a game state, such as `Maxie` versus `Minnie`, without needing access to the full abstract state.

For sequences we have extended the `SEQUENCE` signature with the following components to enable viewing a sequence as a list:

```
datatype 'a lview = Nil | Cons of 'a * 'a seq
val showl : 'a seq -> 'a lview
val hidel : 'a lview -> 'a seq

(* ENSURES:  showl (hidel v) ≅ v, hidel (showl s) ≅ s.                    *)
(* ENSURES:  showl s ≅ Nil if s is an empty sequence.                     *)
(* ENSURES:  showl s ≅ Cons(v, s') if nth s 0 ≅ v and s' is the tail of s. *)
```

Because the datatype definition is in the signature, the constructors can be used outside the abstraction boundary. The `showl` and `hidel` functions convert between sequences and list views. Here is an example of using this view to perform list-like pattern matching:

```
case Seq.showl s of
    Seq.Nil => ...
  | Seq.Cons (x, s') => ... uses x and s' ...
```

`lview` exposes that a sequence is either empty or has a first element and a rest (or tail). The tail is another *sequence*, not an `lview`—for efficiency, we don't want to convert the whole sequence to a list just to peek at the first element. Thus, `showl` lets you do one level of pattern matching at a time: you can write patterns like `Seq.Cons(x,s')` but not `Seq.Cons(x,Seq.Nil)`.

We have also provided `hidel`, which converts a view back to a sequence.

Note that `Seq.hidel(Seq.Cons(x,s'))` is equivalent to `(Seq.cons x s')`.
Similarly, `Seq.hidel(Seq.Nil)` is equivalent to `Seq.empty()`. This relationship may be used as the basis for an induction principle for sequences, enabling us to reason about sequences inductively, as if they were lists. For example:

> Let `t` be a type and `P: t Seq.seq -> bool` be a predicate.
> To prove "For all `s : t Seq.seq`, `P(s)`", it suffices to show
>
> (a) `P(Seq.hidel(Seq.Nil))`
>
> (b) For all `x : t` and `s' : t Seq.seq`,
> if `P(s')` then `P(Seq.hidel(Seq.Cons(x,s')))`

We have also endowed `SEQUENCE` with a tree view that lets one pattern match on a sequence as if it were a tree.

```
datatype 'a tview = Empty | Leaf of 'a | Node of 'a seq * 'a seq
val showt : 'a seq -> 'a tview
val hidet : 'a tview -> 'a seq
```