Simulating Logical Devices

The key innovation in the study of computation was the development of machines for the mechanization of algorithms. We have already seen Turing's proposal for such a device; however, we have not seen any relationship between his and Church's theories. In this chapter we will compose with our symbolic language a simulation of different logical devices.

Turing Machines

Having built up our language to a point of high-level abstraction, we will now try to simulate a trivial computational platform, a Turing Machine. In doing so we will address many key issues like immutability, hash-tables, and, once again, recursion. Additionally, we will become comfortable with the idea of interpreting one platform within another; this ability to interpret is the key to abstraction in computation.

A Ruleset

A Turing Machine computes values based on an initial state, initial values, and rules of transition from a given state and value to a new state and value. The machine itself has a tape full of values and a head which navigates the tape, moving either right of left one slot at any given time. Additionally, this head maintains an idea of state, the mode in which it is observing a given value.

Together these primitive capabilities are enough to compute any algorithm. The usual manipulation could be broken down into a series of steps, most likely represented as different states. Each state, in turn, maintains a specific array of ways in which to manipulate read values, and in which direction to move in each case.

All of this methodology is governed by a single ruleset. Hence to simulate one, we will undoubtedly need a representation of these rules. The following is one way, and the way which we will choose, of representing such a set. We have already seen use of lists as hash-tables, so this should not be a surprising design decision.

```
(let
  rules
  '(((A 0) (1 R H))
    ((A 1) (0 R H)))
  ...)
```

A ruleset like the one above serves to tell a simulation in what way to behave given a certain input state. Hence together with our earlier defined `assoc` function and an actual executor of the matching behavior, this ruleset will handle all state logic.

Fundamental States

In order for our simulation to ever end we will need to designate a specific state the *halt-state*. In our implementation, `H` will signal the end of an algorithm. Additionally, the state in which our machine is initialized will not be the choice of the ruleset, and so we choose to once again arbitrarily designate a specific state value for this case. The state named `A` will be the initialization state of all simulations.

Given these determined special states, we will need to set the initial state and provide checking for the halt state. Hence our recursive, rule-applying function needs to accept a state, as well as head position, rules, state, and tape values, and to check whether it is in the halt state. If this is not the case,

it will then apply the current rule and repeat.

Mutability

The Lambda Calculus does not allow for mutation of values, thus we will need to model this behavior by maintaining a modified value upon each change, one that is the response of a given function. Let's work out an example of working around immutability in lists.

```
(set
  (lambda
    (key val hash)
    (map
      (lambda
        (item)
        (if
          (equal? (car item) key)
          (cons key (cons val))
          item))
      hash)))
```

In this case we utilized `map` to cycle through the items of the hash. Within the map we substituted for the value with the specified key. In our writing to the tape, we will need to use similar tactics.

A Turing Machine writes a value to a specific slot of its tape, namely, the slot upon which the head is currently resting. Within our simulator, this slot is specified by the current index of the simulation. Hence, the writing function we will require is one designed to write to a specific index of a tape.

In the following function definition, we recurse with the tail of the list until we reach the specified index, and we then perform the substitution, ending recursion.

```
(letrec write-rule
  (lambda (write-rule tape index rule)
  (if
    (null? tape)
    tape
    (if
      (equal? index 0)
      (cons (car rule) (cdr tape))
      (cons (car tape) (write-rule (cdr tape) (- index 1) rule))))))
```

Above, we defined `write-rule` as a recursive function accepting a tape, index, and rule. If we have run out of tape, we return the empty tape; however, if we have tape left we take one of two paths in evaluation. If the index to which we wish to write is zero, we return the rule-specified value embedded into the list where the first item previously would have resided. If, however, we are writing to another index, we simply reduce the problem to writing to the item one less in index on tape, excluding the first item.

The Event Loop

The simulation is based on the idea of following rule to rule until the algorithm terminates. Hence rules are executed recursively until the halt-state is reached. At this point, a final table is returned. The iteration function is defined below.

```
(letrec iterate (lambda
    (iterate index rules state tape)
    (if
        (equal? state 'H)
        tape
        (iterate-rule
            iterate
            (cadr (assoc (list state index) rules))
            rules
            index
            tape)))
```

The `iterate` function definition above utilized `letrec` to receive itself as an argument. Its use of this value is subtly different from our past use cases. In `iterate`, we pass the function itself as an argument to another function, `iterate-rule`. For this reason, we can call `iterate` and `iterate-rule` *mutually recursive*, that is, because `iterate` invokes `iterate-rule` and `iterate-rule` in turn invokes `iterate`.

There are multiple dependencies to the above function which we have not yet defined. In the following section we will put them all together with the `iterate` function and achieve our final goal of simulation.

A Simulator

All of the above principles can be combined to form a Turing Machine simulator. The definition of `iterate` is dependent upon a few helper functions. First of all, there are a couple very basic shorthands which are defined below.

```
(cadr          (lambda (x) (car (cdr x))))
(caddr         (lambda (x) (car (cdr (cdr x)))))
```

Now we move on to the functions provided for applying a rule and for applying a shift in the head. To shift the index we simply handle the case of right motion, i.e., `'R` direction as an upward shift, as well as any other cases.

```
(move (lambda
    (index motion)
    (if
        (equal? motion 'R)
        (+ 1 index)
        (- 1 index))))
```

The definition of `move` above is very simple in nature. A current index and direction of motion are received as argument, and a new index is then returned. If the direction is `'R` then the index will increase, but if it is not, i.e., if it is `'L`, it will decrease. Furthermore, keeping in mind the earlier definition of subtraction based on Lambda Calculus primitives, you will recall that subtracting one from zero will result in zero. This behavior is convenient in this case, avoid strange edge-case behavior.

Example usage of the `move` function would be as follows.

```
(move 5 'R)      ⟹ 6
(move 3 'L)      ⟹ 2
```

Now we move on to a prior utilized function for the application of a rule to the tape. The rule applier receives the earlier defined `iterate` function as an argument, and then applies it to the moved index, the ruleset, the rule-provided state, and the new tape.

```
(iterate-rule (lambda
   (iterate rule rules index tape)
   (iterate
      (move index (cadr rule))
      rules
      (caddr rule)
      (write-rule tape index rule))))
```

This function consists only of basic manipulations of the rule to parse out the modifications needing to be applied. With all of the dependencies defined, we achieve the following comprehensive definition of a Turing Machine simulator.

```
(let *
   ((index 0)
    (rules '(((A 0) (1 R H))))
    (state 'A)
    (cadr (…))
    (caddr (…))
    (move (lambda
       (index motion)
       (…)
    (iterate-rule (lambda
       (iterate rule rules index tape)
       (…)
   (letrec write-rule (lambda
       (write-rule tape index rule)
       (…)
       (letrec iterate (lambda
          (iterate index rules state tape)
          (…)
   (write (iterate index rules state '(0 0 0))))))
```

Computation with Turing Machines

A half-adder as a Turing Ruleset would look like the following.

```
(let
   rules
   '(((A 0) (0 R Z))
      ((A 1) (1 R C))
      ((Z 0) (0 R H))
      ((Z 1) (1 L N))
      ((C 0) (1 L N))
      ((C 1) (0 L Y))
      ((N 0) (0 R H))
      ((N 1) (0 R H))
      ((Y 0) (1 R H))
      ((Y 1) (1 R H)))
   (write (iterate 0 rules 'A '(0 0 0))))
```

Circuits

Circuits are quite different in nature from the previously discussed Turing Machine. The main reason for this difference is that components, analogous to the prior discussed rules, are dependent directly upon each other, while in a Turing Machine a single processing unit handled transitions between states.

Structure of Circuits

Our model of circuits will consist of two component types, (a) relational boxes, and (b) wires. Relational boxes are atomic relations between circuit values, accepting input as electrical signals and outputting an electrical signal. Wires serve to connect these boxes to each other, bearing these electrical signals in one of two sates. A wire bearing current is said to have the boolean value true

(`#t`), but a wire without current is said to be of the boolean value false (`#f`).

Now, given these ideas of relational boxes and wires, we add abstraction to reach a view of relational boxes as logical primitives. For example, there may be a relational box named `gateA` which accepts a single wire and maps to the following specified outputs.

$$\begin{aligned}(gateA\ \#\mathrm{t}) &\Longrightarrow \#\mathrm{f}\\ (gateA\ \#\mathrm{f}) &\Longrightarrow \#\mathrm{t}\end{aligned}$$

Of course, this sort of truth table maintains an electrical interpretation. Specifically, such a relational box would output current when receiving no current, but would output no current if receiving current.

The relational boxes we will take as primitive are similar to our boolean operators already defined. We utilize an `and-gate`, an `or-gate`, and a `not-gate`. The first two gates accept two wires as input values and output to another wire a current value based on their logical operation. Hence an `and-gate` accepting two current-bearing wires will direct current to its output wire. A `not-gate` on the other hand accepts a single wire for input value and outputs the opposite value to another wire.

To begin our design of circuits, we design a relation constituent of the boolean operators listed above as primitives.

```
(half-adder
  (lambda (a b)
  (let *
    ((s (and (or a b) (not (and a b))))
     (r (and a b)))
    (cons r (cons s nil)))))
```

The above procedure is known as a half-adder. Given two bits as input, this procedure determines the added value, including any carried value. Recall that our language was designed not only for evaluation by machine, but for representation of ideas like the one presented above.

The methodology of the half-adder should be for the most part apparent. Given input values named `a` and `b`, output values named `s` and `r` need to be determined. `s` is true when one, but not both, of the inputs is true, and `r` when both are true. Hence `s` represents the first digit of a binary result, and `r` the second or carried value.

Let's look at some examples of the behavior of a half-adder.

$$\begin{aligned}(\text{half-adder}\ \#\mathrm{f}\ \#\mathrm{f}) &\Longrightarrow {}'(\#\mathrm{f}\ \#\mathrm{f})\\ (\text{half-adder}\ \#\mathrm{f}\ \#\mathrm{t}) &\Longrightarrow {}'(\#\mathrm{f}\ \#\mathrm{t})\\ (\text{half-adder}\ \#\mathrm{t}\ \#\mathrm{f}) &\Longrightarrow {}'(\#\mathrm{f}\ \#\mathrm{t})\\ (\text{half-adder}\ \#\mathrm{t}\ \#\mathrm{t}) &\Longrightarrow {}'(\#\mathrm{t}\ \#\mathrm{f})\end{aligned}$$

If you are not familiar with the behavior of binary digits when adding, note that the above examples exhibit the basics of this behavior. If we were to represent current instead by either `1` or `0`, we would achieve the following, more clearly binary, behavior.

$$\begin{aligned}(\text{half-adder}\ 0\ 0) &\Longrightarrow {}'(0\ 0)\\ (\text{half-adder}\ 0\ 1) &\Longrightarrow {}'(0\ 1)\\ (\text{half-adder}\ 1\ 0) &\Longrightarrow {}'(0\ 1)\\ (\text{half-adder}\ 1\ 1) &\Longrightarrow {}'(1\ 0)\end{aligned}$$

Hopefully this example has helped to illustrate the emergence of relatively high-level ideas like arithmetic from basic controlled flow of current. In the following sections we will attempt to depart from a purely boolean-arithmetic driven outlook on circuits toward a generic circuit structure definition process

and simulator.

Inter-Dependency

In our presentation of circuit design, we utilized `let*` to display relations in order of their dependency. However, you will notice that we repeated some computations without separating them out, and more importantly that all computed values were statically set to a primitive manipulation, never to be changed. Hence, our prior definition does not accurately model an actual circuit; currents cannot be updated and changes cannot propagate.

To better reflect the reality of circuit design, we will allow a circuit structure to be defined *holistically* and *symbolically*, and for this structure to be utilized to compute the values of individual components. To say that our structure will be defined holistically means that the entire circuit blueprint may be laid prior to any computation, which brings us to our next point. Since definition will be symbolic, using names rather than values, our relations can be established between yet-to-be-defined values.

We will use a table like those which we have already seen for our basic data-structure. The table will be built up with named components, each being manipulations of the circuit. We begin with a basic realization of this idea.

```
(get-gate
   (lambda
      (name env)
      (assoc name env)))
(set-gate
   (lambda
      (name value env)
      (set name value env)))
```

The above definitions are simply aliases to the `assoc` and `set` functions of regular hash-tables. We have yet to implement the idea of gates as manipulations of the circuit. In order to do this, we will need a clear means of applying a manipulation to a given object. This idea is key to an object-oriented outlook on programming which we will discuss in the following.

Methods on Objects

There is a paradigm in programming known as object oriented programming. Under this methodology, everything is an object containing data and behavior, *values* and *methods*. We have already seen the functional architecture for associated values, that is, a table or list of pairs. However, the key to this object-oriented approach is that the methods are not regular functions but, rather, maintain a context in which they operate. These methods of an object should operate upon the object itself. To simulate this idea in our language, we will make all methods a function of the object in which they exist. Hence we can define a generic method applier as follows.

```
(method
   (lambda
      (obj mname)
      ((assoc mname obj) obj)))
```

Usage of this convenience function would then look like the following, applying a named function to an object.

```
(method person 'greet)
```

In this case `person` serves as an object, and `greet` as a named method on that object. What does it mean to be an object? In the case of our implementation, an object is merely a data-structure, like any other table;

however, the distinctive trait is the inclusion of methods. Methods allow for the coupling of functions to a specific data-set. In our example above, a greet function is attached to the `person`, and easily called by name to perform some action in the specific context of the `person` object.

We will utilize the concepts of object-oriented programming (OOP) in our design of gates. A gate will be a table containing some values and some methods. The only value will be named `value`, the boolean value of the gate, and the methods will be named `get` and `set`, performing the manipulations of the value which their names would suggest. Hence, we would have a basic constructor of a gate like the following.

```
(make-gate
   (lambda
      (value get set)
      (list
         (list 'value value nil)
         (list 'get get nil)
         (list 'set set nil)
         nil)))
```

Given this structure, we redefine our gate getter and setter to simplify interfacing with this structure as follows.

```
(get-gate
   (lambda
      (name env)
      (let
         gate (assoc name env)
         ((assoc 'get gate) gate env))))
(set-gate
   (lambda
      (name value env)
      (set name value env)))
```

Notice that the main change was in making the `get-gate` function get the boolean value of a gate rather than the gate itself. `set-gate` remained a function returning the mutated environment, for the sake of setting up a circuit initially.

Child Object Definitions

A *child* of an object is one inheriting the structure of its parent and either restricting or expanding the construction, value or method interfaces. The following is a child of the gate definition for constant-value gates.

```
(const-gate
   (lambda
      (value)
      (make-gate
         value
         (lambda (obj env) (assoc 'value obj))
         (lambda (obj value) (set 'value value obj)))))
```

As you were made to expect in our discussion of object-oriented programming, both of the methods accept as their first parameter the gate

object itself. The getter and setter are very simple, based around the value attached to the gate object. Building upon this simple architecture, we will define a generic relational gate object.

```
(fn-gate
  (lambda
    (fn a b)
    (make-gate
      (lambda (a b) (fn a b))
      (lambda (obj env)
        ((assoc 'value obj)
         (get-gate a env)
         (get-gate b env))
      (lambda (obj value) obj))))
```

The above definition makes the value of the gate a method as well. The getter then applies the `value` method to the `get`-wrapped values of the two input components. The relation tying together these input components is passed as the first argument to the `fn-gate` constructor. This is to say that when getting the value of an `fn-gate`, the values upon which it depends will be gotten as well in a sort of cascading dependency. These dependencies will then be assessed based on the function specific to that instance of `fn-gate`, maybe logical or, for example.

We now define, in turn, children of the `fn-gate` constructor easily as follows.

```
(or-gate      (fn-gate (lambda (a b) (or a b))))
(and-gate     (fn-gate (lambda (a b) (or a b))))
(not-gate   (fn-gate (lambda (a b) (not b)) #f))
```

A Simulator

Putting together all prior defined functions we have the following.

```
(let *
  ((pairing
     (lambda (a b) (…)))
   (make-gate
     (lambda (value get set) (…)))
   (const-gate
     (lambda (value) (…)))
   (get-gate
     (lambda (name env) (…)))
   (set-gate
     (lambda (name value env) (…)))
   (fn-gate
     (lambda (fn a b) (…)))
   (or-gate (fn-gate (lambda (a b) (…))))
   (and-gate (fn-gate (lambda (a b) (…))))
   (not-gate (fn-gate (lambda (a b) (…)) #f)))
  (…))
```

Computation with Circuits

A half-adder utilizing our simulator would look like the following.

```
(let*
    ((env (set-gate 'a (const-gate #t) env))
     (env (set-gate 'b (const-gate #t) env))
     (env (set-gate '1 (or-gate 'a 'b) env))
     (env (set-gate '2 (and-gate 'a 'b) env))
     (env (set-gate '3 (not-gate '2) env))
     (env (set-gate '4 (and-gate '1 '2) env)))
    (cons (get-gate '4 env) (cons (get-gate '2 env) nil)))
```

by Matt Neary