18 | Imitation Learning

Programming is a skill best acquired by practice and example rather than from books. – Alan Turing

Learning Objectives:

- Be able to formulate imitation learning problems.
- Understand the failure cases of simple classification approaches to imitation learning.
- Implement solutions to those problems based on either classification or dataset aggregation.
- Relate structured prediction and imitation learning.

Dependencies:

So far, we have largely considered machine learning problems in which the goal of the learning algorithm is to make a *single* prediction. In many real world problems, however, an algorithm must make a *sequence* of decisions, with the world possibly changing during that sequence. Such problems are often called **sequential decision making** problems. A straightforward example—which will be the running example for this chapter—is that of self-driving cars. We want to train a machine learning algorithm to drive a car. But driving a car is not a single prediction: it's a sequence of predictions over time. And as the machine is driving the car, the world around it is changing, often based on its own behavior. This creates complicated feedback loops, and one of the major challenges we will face is how to deal with these feedback loops.

To make this discussion more concrete, let's consider the case of a self-driving car. And let's consider a very simplistic car, in which the only decision that has to be made is how to steer, and that's between one of three options: $\{\text{left}, \text{right}, \text{none}\}$. In the imitation learning setting, we assume that we have access to an **expert** or **oracle** that already knows how to drive. We want to watch the expert driving, and learn to imitate their behavior. Hence: **imitation learning** (sometimes called **learning by demonstration** or **programming by example**, in the sense that programs are learned, and not implemented).

At each point in time $t = 1 \dots T$, the car recieves sensor information $\boldsymbol{x}_t$ (for instance, a camera photo ahead of the car, or radar readings). It then has to take an action, a_t; in the case of the car, this is one of the three available steering actions. The car then suffers some loss ℓ_t; this might be zero in the case that it's driving well, or large in the case that it crashes. The world then changes, moves to time step $t + 1$, sensor readings $\boldsymbol{x}_{t+1}$ are observed, action a_{t+1} is taken, loss ℓ_{t+1} is suffered, and the process continues.

The goal is to learn a function f that maps from sensor readings $\boldsymbol{x}_t$ to actions. Because of close connections to the field of **reinforcement learning**, this function is typically called a **policy**. The measure of success of a policy is: if we were to run this policy, how much total loss would be suffered. In particular, suppose that the **trajectory** (denoted $\boldsymbol{\tau}$) of observation/action/reward triples encountered by your policy is:

$$\boldsymbol{\tau} = \boldsymbol{x}_1\,,\ \underbrace{a_1}_{=f(\boldsymbol{x}_1)},\ell_1\,,\ \boldsymbol{x}_2\,,\ \underbrace{a_2}_{=f(\boldsymbol{x}_2)},\ell_2\,,\ \dots\,,\ \boldsymbol{x}_T\,,\ \underbrace{a_T}_{=f(\boldsymbol{x}_T)},\ell_T \tag{18.1}$$

The losses ℓ_t depend implicitly on the state of the world and the actions of the policy. The goal of f is to minimize the expected loss $\mathbb{E}_{\boldsymbol{\tau}\sim f}\left[\sum_{t=1}^{T}\ell_t\right]$, where the expectation is taken over all randomness in the world, and the sequence of actions taken is according to f.[1]

[1] It's completely okay for f to look at more than just $\boldsymbol{x}_t$ when making predictions; for instance, it might want to look at $\boldsymbol{x}_{t-1}$, or a_{t-1} and a_{t-2}. As long as it only references information from the *past*, this is fine. For notational simplicity, we will assume that all of this relevant information is summarized in $\boldsymbol{x}_t$.

18.1 Imitation Learning by Classification

We will begin with a straightforward, but brittle, approach to imitation learning. We assume access to a set of *training trajectories* taken by an expert. For example, consider a self-driving car, like that in Figure 18.1. A single trajectory $\boldsymbol{\tau}$ consists of a sequence of observations (what is seen from the car's sensors) and a sequence of actions (what action did the expect take at that point in time). The idea in imitation learning by classification is to learn a classifier that attempts to mimic the expert's action based on the observations at that time.

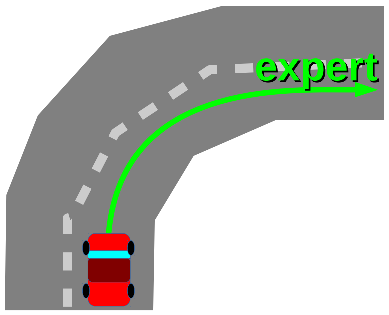

Figure 18.1: A single expert trajectory in a self-driving car.

In particular, we have $\boldsymbol{\tau}_1, \boldsymbol{\tau}_2, \dots, \boldsymbol{\tau}_N$. Each of the N trajectories comprises a sequence of T-many observation/action/loss triples, where the action is the action taken by the expert. T, the length of the trajectory is typically called the **time horizon** (or just **horizon**). For instance, we may ask an expert human driver to drive $N = 20$ different routes, and record the observations and actions that driver saw and took during those routes. These are our training trajectories. We assume for simplicity that each of these trajectories is of fixed length T, though this is mostly for notational convenience.

The most straightforward thing we can do is convert this expert data into a big multiclass classification problem. We take our favorite multiclass classification algorithm, and use it to learn a mapping from $\boldsymbol{x}$ to a. The data on which it is trained is the set of all observation/action pairs visited during any of the N trajectories. In total, this would be NT examples. This approach is summarized in Algorithm 18.1 for training and Algorithm 18.1 for prediction.

How well does this approach work?

The first question is: how good is the expert? If we learn to mimic an expert, but the expert is no good, we lose. In general, it also seems unrealistic to believe this algorithm should be able to *improve* on the expert. Similarly, if our multiclass classification algorithm $\mathcal{A}$ is crummy, we also lose. So part of the question "how well does

Algorithm 43 SupervisedImitationTrain($\mathcal{A}, \tau_1, \tau_2, \dots, \tau_N$)

```
1: $D \leftarrow \langle (x, a) \ : \ \forall n \ , \ \forall (x, a, \ell) \in \tau_n \rangle$        // collect all observation/action pairs
2: return $\mathcal{A}(D)$                                          // train multiclass classifier on D
```

Algorithm 44 SupervisedImitationTest(f)

```
1: for $t = 1 \dots T$ do
2:     $x_t \leftarrow$ current observation
3:     $a_t \leftarrow f(x_t)$                                    // ask policy to choose an action
4:     take action $a_t$
5:     $\ell_t \leftarrow$ observe instantaneous loss
6: end for
7: return $\sum_{t=1}^{T} \ell_t$                                     // return total loss
```

this work" is the more basic question of: what are we even trying to measure?

There is a nice theorem[2] that gives an upper bound on the loss suffered by the SupervisedIL algorithm (Algorithm 18.1) as a function of (a) the quality of the expert, and (b) the error rate of the learned classifier. To be clear, we need to distinguish between the loss of the policy when run for T steps to form a full trajectory, and the error rate of the learned classifier, which is just it's average multiclass classification error. The theorem states, roughly, that the loss of the learned *policy* is at most the loss of the expert plus T^2 times the error rate of the classifier.

[2] Ross et al. 2011

Theorem 18 (Loss of SupervisedIL)**.** *Suppose that one runs Algorithm 18.1 using a multiclass classifier that optimizes the 0-1 loss (or an upperbound thereof). Let ϵ be the error rate of the underlying classifier (in expectation) and assume that all instantaneous losses are in the range $[0, \ell^{(max)}]$. Let f be the learned policy; then:*

$$\underbrace{\mathbb{E}_{\tau \sim f}\left[\sum_t \ell_t\right]}_{\text{loss of learned policy}} \leq \underbrace{\mathbb{E}_{\tau \sim expert}\left[\sum_t \ell_t\right]}_{\text{loss of expert}} + \ell^{(max)} T^2 \epsilon \qquad (18.2)$$

Intuitively, this bound on the loss is about a factor of T away from what we might hope for. In particular, the multiclass classifier makes errors on an ϵ fraction of it's actions, measured by zero/one loss. In the worst case, this will lead to a loss of $\ell^{(max)}\epsilon$ for a single step. Summing all these errors over the entire trajectory would lead to a loss on the order of $\ell^{(max)} T\epsilon$, which is a factor T better than this theorem provides. A natural question (addressed in the next section) is whether this is analysis is tight. A related question (addressed in the section after that) is whether we can do better. Before getting there, though, it's worth highlighting that an extra factor of T is *really* *bad.* It can cause even very small multiclass error rates to blow up; in particular, if $\epsilon \geq 1/T$, we lose, and T can be in the hundreds or more.

18.2 Failure Analysis

The biggest single issue with the supervised learning approach to imitation learning is that it cannot learn to recover from failures. That is: it has only been trained based on expert trajectories. This means that the only training data it has seen is that of an expert driver. If it *ever* veers from that state distribution, it may have no idea how to recover. As a concrete example, perhaps the expert driver never ever gets themselves into a state where they are directly facing a wall. Moreover, the expert driver probably tends to drive forward more than backward. If the imperfect learner manages to make a few errors and get stuck next to a wall, it's likely to resort to the general "drive forward" rule and stay there forever. This is the problem of **compounding error**; and yes, it does happen in practice.

It turns out that it's possible to construct an imitation learning problem on which the T^2 compounding error is unavoidable. Consider the following somewhat artificial problem. At time $t = 1$ you're shown a picture of either a zero or a one. You have two possible actions: press a button marked "zero" or press a button marked "one." The "correct" thing to do at $t = 1$ is to press the button that corresponds to the image you've been shown. Pressing the correct button leads to $\ell_1 = 0$; the incorrect leads to $\ell_1 = 1$. Now, at time $t = 2$ you are shown another image, again of a zero or one. The correct thing to do in this time step is the xor of (a) the number written on the picture you see right now, and (b) the correct answer from the previous time step. This holds in general for $t > 1$.

There are two important things about this construction. The first is that the expert can easily get zero loss. The second is that once the learned policy makes a single mistake, this can cause it to make *all* future decisions incorrectly. (At least until it "luckily" makes another "mistake" to get it back on track.)

Based on this construction, you can show the following theorem[3].

[3] Kääriäinen 2006

Theorem 19 (Lower Bound for SupervisedIL). *There exist imitation learning problems on which Algorithm 18.1 is able to achieve small classification error $\epsilon \in [0, 1/T]$ under an optimal expert, but for which the test loss is lower bounded as:*

$$\underbrace{\mathbb{E}_{\tau \sim f}\left[\sum_t \ell_t\right]}_{\text{loss of learned policy}} \geq \frac{T+1}{2} - \frac{1}{4\epsilon}\left[1 - (1-2\epsilon)^{T+1}\right] \tag{18.3}$$

which is bounded by $T^2\epsilon$ and, for small ϵ, grows like $T^2\epsilon$.

Up to constants, this gives matching upper and lower bounds for the loss of a policy learned by supervised imitation learning that is pretty far (a factor of T) from what we might hope for.

18.3 Dataset Aggregation

Supervised imitation learning fails because once it gets "off the expert path," things can go really badly. Ideally, we might want to train our policy to deal with *any* possible situation it could encounter. Unfortunately, this is unrealistic: we cannot hope to be able to train on every possible configuration of the world; and if we could, we wouldn't really need to learn anyway, we could just memorize. So we want to train f on a subset of world configurations, but using "configurations visited by the expert" fails because f cannot learn to recover from its own errors. Somehow what we'd like to do is train f to do well on the configurations that it, itself, encounters!

This is a classic chicken-and-egg problem. We want a policy f that does well in a bunch of world configurations. What set of configurations? The configurations that f encounters! A very classic approach to solving chicken-and-egg problems is iteration. Start with some policy f. Run f and see what configurations is visits. Train a new f to do well there. Repeat.

This is exactly what the Dataset Aggregation algorithm ("Dagger") does. Continuing with the self-driving car analogy, we first let a human expert drive a car for a while, and learn an initial policy f_0 by running standard supervised imitation learning (Algorithm 18.1) on the trajectories visited by the human. We then do something unusual. We put the human expert in the car, and record their actions, but the car behaves not according to the expert's behavior, but according to f_0. That is, f_0 is in control of the car, and the expert is trying to steer, but the car is ignoring them[4] and simply recording their actions as training data. This is shown in Figure 18.2.

[4] This is possibly terrifying for the expert!

Based on trajectories generated by f_0 but actions given by the expert, we generate a new dataset that contains information about how to recover from the errors of f_0. We now will train a new policy, f_1. Because we don't want f_1 to "forget" what f_0 already knows, f_1 is trained on the union of the initial expert-only trajectories together with the new trajectories generated by f_0. We repeat this process a number of times MaxIter, yielding Algorithm 18.3.

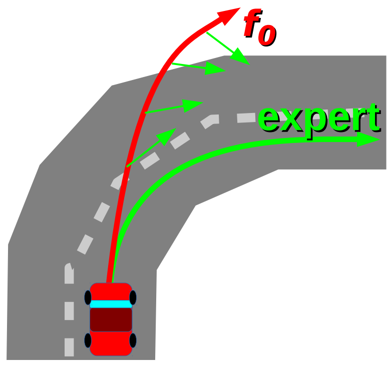

Figure 18.2: In DAgger, the trajectory (red) is generated according to the previously learned policy, f_0, but the gold standard actions are given by the expert.

This algorithm returns the list of *all* policies generated during its run. A very practical question is: which one should you use? There are essentially two choices. The first choice is just to use the final policy learned. The problem with this approach is that Dagger can be somewhat unstable in practice, and policies do not monotonically

Algorithm 45 DaggerTrain($\mathcal{A}$, MaxIter, N, expert)

```
1: $\langle \boldsymbol{\tau}_n^{(0)} \rangle_{n=1}^N \leftarrow$ run the expert $N$ many times
2: $D_0 \leftarrow \big\langle (\boldsymbol{x}, a) : \forall n, \forall(\boldsymbol{x}, a, \ell) \in \boldsymbol{\tau}_n^{(0)} \big\rangle$   // collect all pairs (same as supervised)
3: $f_0 \leftarrow \mathcal{A}(D_0)$   // train initial policy (multiclass classifier) on $D_0$
4: for $i = 1 \dots MaxIter$ do
5:     $\langle \boldsymbol{\tau}_n^{(i)} \rangle_{n=1}^N \leftarrow$ run policy $f_{i-1}$ $N$-many times   // trajectories by $f_{i-1}$
6:     $D_i \leftarrow \big\langle (\boldsymbol{x}, \text{expert}(\boldsymbol{x})) : \forall n, \forall(\boldsymbol{x}, a, \ell) \in \boldsymbol{\tau}_n^{(i)} \big\rangle$   // collect data set
                                                                          // observations $\boldsymbol{x}$ visited by $f_{i-1}$
                                                                          // but actions according to the expert
7:     $f_i \leftarrow \mathcal{A}\left(\bigcup_{j=0}^{i} D_j\right)$   // train policy $f_i$ on union of all data so far
8: end for
9: return $\langle f_0, f_1, \dots, f_{\text{MaxIter}} \rangle$   // return collection of all learned policies
```

improve. A safer alternative (as we'll see by theory below) is to test all of them on some held-out "development" tasks, and pick the one that does best there. This requires a bit more computation, but is a much better approach in general.

One major difference in *requirements* between Dagger (Algorithm 18.3) and SupervisedIL (Algorithm 18.1) is the requirement of interaction with the expert. In SupervisedIL, you only need access to a bunch of trajectories taken by the expert, *passively*. In Dagger, you need access to them expert themselves, so you can ask questions like "if you saw configuration $\boldsymbol{x}$, what would you do?" This puts *much* more demand on the expert.

Another question that arises is: what should N, the number of trajectories generated in each round, be? In practice, the initial N should probably be reasonably large, so that the initial policy f_0 is pretty good. The number of trajectories generated by iteration subsequently can be much smaller, perhaps even just one.

Intuitively, Dagger should be less sensitive to compounding error than SupervisedIL, precisely because it gets trained on observations that it is likely to see at test time. This is formalized in the following theorem:

Theorem 20 (Loss of Dagger)**.** *Suppose that one runs Algorithm 18.3 using a multiclass classifier that optimizes the 0-1 loss (or an upperbound thereof). Let ϵ be the error rate of the underlying classifier (in expectation) and assume that all instantaneous losses are in the range $[0, \ell^{(max)}]$. Let f be the learned policy; then:*

$$\underbrace{\mathbb{E}_{\boldsymbol{\tau}\sim f}\left[\sum_t \ell_t\right]}_{\text{loss of learned policy}} \leq \underbrace{\mathbb{E}_{\boldsymbol{\tau}\sim expert}\left[\sum_t \ell_t\right]}_{\text{loss of expert}} + \ell^{(max)} T \epsilon + O\left(\frac{\ell^{(max)} T \log T}{MaxIter}\right) \tag{18.4}$$

Furthermore, if the loss function is strongly convex in f, and MaxIter is

$\tilde{O}(T/\epsilon)$, *then:*

$$\underbrace{\mathbb{E}_{\boldsymbol{\tau}\sim f}\left[\sum_t \ell_t\right]}_{\text{loss of learned policy}} \leq \underbrace{\mathbb{E}_{\boldsymbol{\tau}\sim expert}\left[\sum_t \ell_t\right]}_{\text{loss of expert}} + \ell^{(max)}T\epsilon + O(\epsilon) \tag{18.5}$$

Both of these results show that, assuming MaxIter is large enough, the loss of the learned policy f (here, taken to be the best on of all the MaxIter policies learned) grows like $T\epsilon$, which is what we hope for. Note that the final term in the first bound gets small so long as MaxIter is at least $T \log T$.

18.4 Expensive Algorithms as Experts

Because of the strong requirement on the expert in Dagger (i.e., that you need to be able to query it many times during training), one of the most substantial use cases for Dagger is to learn to (quickly) imitate otherwise slow algorithms. Here are two prototypical examples:

1. Game playing. When a game (like chess or minecraft) can be run in simulation, you can often explicitly compute a semi-optimal expert behavior with brute-force search. But this search might be too computationally expensive to play in real time, so you can use it during training time, learning a fast policy that attempts to mimic the expensive search. This learned policy can then be applied at test time.

2. Discrete optimizers. Many discrete optimization problems can be computationally expensive to run in real time; for instance, even shortest path search on a large graph can be too slow for real time use. We can compute shortest paths offline as "training data" and then use imitation learning to try to build shortest path optimizers that will run sufficiently efficiently in real time.

Consider the game playing example, and for concreteness, suppose you are trying to learn to play solitaire (this is an easier example because it's a single player game). When running DaggerTrain (Algorithm 18.3 to learn a chess-playing policy, the algorithm will repeatedly ask for expert($\boldsymbol{x}$), where $\boldsymbol{x}$ is the current state of the game. What should this function return? Ideally, it should return the/an action a such that, if a is taken, and then the rest of the game is played optimally, the player wins. Computing this exactly is going to be very difficult for anything except the simplest games, so we need to restort to an approxiamtion.

Algorithm 46 DepthLimitedDFS($x, h,$ MaxDepth)

```
if $x$ is a terminal state or MaxDepth $\leq 0$ then
    return $(\perp, h(x))$                                  // if we cannot search deeper
                        // return "no action" ($\perp$) and the current heuristic score
else
    $BestAction, BestScore \leftarrow \perp, -\infty$       // keep track of best action & its score
    for all actions $a$ from $x$ do
        $(\_, score) \leftarrow$ DepthLimitedDFS($x \circ a, h,$ MaxDepth $- 1$)   // get score
                        // for action $a$, depth reduced by one by appending $a$ to $x$
        if $score >$ BestScore then
            $BestAction, BestScore \leftarrow a, score$     // update tracked best action & score
        end if
    end for
end if
return (BestAction, BestScore)                              // return best found action and its score
```

A common strategy is to run a depth-limited depth first search, starting at state x, and terminating after at most three of four moves (see Figure 18.3). This will generate a search tree. Unless you are very near the end of the game, none of the leaves of this tree will correspond to the end of the game. So you'll need some heuristic, h, for evaluating states that are non-terminals. You can propagate this heuristic score up to the root, and choose the action that looks best with this depth four search. This is not necessarily going to be the *optimal* action, and there's a speed/accuracy trade-off for searching deeper, but this is typically effective. This approach summarized in Algorithm 18.4.

Figure 18.3: `imit:dldfs`: Depth limited depth-first search

18.5 Structured Prediction via Imitation Learning

A final case where an expert can often be computed algorithmically arises when one solves structured prediction (see Chapter 17) via imitation learning. It is clearest how this can work in the case of sequence labeling. Recall there that predicted outputs should be *sequences* of labels. The running example from the earlier chapter was:

$$x = \text{" monsters eat tasty bunnies "} \tag{18.6}$$

$$y = \quad \text{noun} \quad \text{verb} \quad \text{adj} \quad \text{noun} \tag{18.7}$$

One can easily cast the prediction of y as a sequential decision making problem, by treating the production of y in a left-to-right manner. In this case, we have a time horizon $T = 4$. We want to learn a policy f that first predicts "noun" then "verb" then "adj" then "noun" on this input.

Let's suppose that the input to f consists of features extracted both from the input ($\boldsymbol{x}$) and the current predicted output prefix $\hat{\boldsymbol{y}}$, denoted $\phi(\boldsymbol{x}, \hat{\boldsymbol{y}})$. For instance, $\phi(\boldsymbol{x}, \hat{\boldsymbol{y}})$ might represent a similar set of features to those use in Chapter 17. It is perhaps easiest to think of f as just a classifier: given some features of the input sentence $\boldsymbol{x}$ ("monsters eat tasty bunnies"), and some features about previous predictions in the output prefix (so far, produced "noun verb"), the goal of f is to predict the tag for the next word ("tasty") in this context.

An important question is: what is the "expert" in this case? Intuitively, the expert should provide the correct next label, but what does this mean? That depends on the loss function being optimized. Under Hamming loss (sum zero/one loss over each individual prediction), the expert is straightforward. When the expert is asked to produce an action for the third word, the expert's response is *always* "adj" (or whatever happens to be the correct label for the third word in the sentence it is currently training on).

More generally, the expert gets to look at $\boldsymbol{x}$, $\boldsymbol{y}$ and a prefix $\hat{\boldsymbol{y}}$ of the output. Note, *importantly*, that the prefix *might be wrong!* In particular, after the first iteration of Dagger, the prefix will be predicted by the learned policy, which may make mistakes! The expert also has some structured loss function ℓ that it is trying to minimize. Like in the previous section, the expert's goal is to choose the action that minimizes the long-term loss according to ℓ on this example.

To be more formal, we need a bit of notation. Let $\text{best}(\ell, \boldsymbol{y}, \hat{\boldsymbol{y}})$ denote the loss (according to ℓ and the ground truth $\boldsymbol{y}$) of the best reachable output starting at $\hat{\boldsymbol{y}}$. For instance, if $\boldsymbol{y}$ is "noun verb adj noun" and $\hat{\boldsymbol{y}}$ is "noun noun", and the loss is Hamming loss, then the best achievable output (predicting left-to-right) is "noun noun adj noun" which has a loss of 1. Thus, best for this situation is 1.

Given that notion of best, the expert is easy to define:

$$\text{expert}(\ell, \boldsymbol{y}, \hat{\boldsymbol{y}}) = \operatorname*{argmin}_{a} \text{best}(\ell, \boldsymbol{y}, \hat{\boldsymbol{y}} \circ a) \tag{18.8}$$

Namely, it is the action that leads to the best possible completion *after* taking that action. So in the example above, the expert action is "adj". For some problems and some loss functions, computing the expert is easy. In particular, for sequence labeling under Hamming loss, it's trivial. In the case that you can compute the expert exactly, it is often called an **oracle**.[5] For some other problems, exactly computing an oracle is computationally expensive or intractable. In those cases, one can often resort to depth limited depth-first-search (Algorithm 18.4) to compute an approximate oracle as an expert.

[5] Some literature calls it a "dynamic oracle", though the extra word is unnecessary.

To be very concrete, a typical implementation of Dagger applied to sequence labeling would go as follows. Each structured training example (a pair of sentence and tag-sequence) gives rise to one trajectory. At training time, a predict tag seqence is generated left-to-right, starting with the empty sequence. At any given time step, you are attempting to predict the label of the tth word in the input. You define a feature vector $\phi(\boldsymbol{x}, \hat{\boldsymbol{y}})$, which will typically consist of: (a) the tth word, (b) left and right neighbors of the tth word, (c) the last few predictions in $\hat{\boldsymbol{y}}$, and (d) anything else you can think of. *In particular,* the features are *not* limited to Markov style features, because we're not longer trying to do dynamic programming. The expert label for the tth word is just the corresponding label in the ground truth $\boldsymbol{y}$. Given all this, one can run Dagger (Algorithm 18.4) exactly as specified.

Moving to structured prediction problems other than sequence labeling problems is beyond the scope of this book. The general framework is to cast your structured prediction problem as a sequential decision making problem. Once you've done that, you need to decide on features (this is the easy part) and an expert (this is often the harder part). However, once you've done so, there are generic libraries for "compiling" your specification down to code.

18.6 Further Reading

TODO further reading