Separating Skills from Preference: Using Learning to Program by Reward

Daniel Shapiro DGS@STANFORD.EDU
Pat Langley LANGLEY@ISLE.ORG
Institute for the Study of Learning and Expertise, 2164 Staunton Court, Palo Alto, CA , 94306 USA

Abstract

Developers of artificial agents commonly take the view that we can only specify agent behavior via the expensive process of implementing new skills. This paper offers an alternative expressed by the *separation* hypothesis: that the behavioral differences among individuals are due to the action of distinct preferences over the same set of skills. We test this hypothesis in a simulated automotive domain by using a reinforcement learning algorithm to induce vehicle control policies, given a structured skill for driving that contains options, and a user-supplied reward function. We show that qualitatively distinct reward functions produce agents with qualitatively distinct behavior over the same set of skills. This leads to a new development metaphor we call Ôprogramming by rewardÕ

1. Motivation and Background

In many domains, humans exhibit complex physical behaviors that let them accomplish sophisticated tasks. Researchers have explored two main approaches to learning such behaviors, each associated with a different class of representational formalisms. One paradigm encodes control knowledge as rules or similar structures (e.g., Laird & Rosenbloom, 1990; Sammut, 1996) that state conditions under which to execute actions. An alternative framework instead specifies some function that maps state-action pairs onto a numeric utility (e.g., Watkins & Dayan, 1992), which is then used to select among actions.

Both approaches have repeatedly demonstrated their ability to learn useful control policies across a broad range of domains, yet each lends itself most naturally to different aspects of intelligent behavior. This idea is best illustrated by work on game playing, where developers regularly use rules or other logical constraints to specify which moves are legal but invoke numeric evaluation functions to select among them. We claim that a similar division of labor will prove useful in research on policies for reactive control, including learning such policies from agent experience.

In this paper, we assume that an agent already has access to a set of logical rules that constrain the allowable actions, but that it must learn the value of its remaining options from delayed reward. Elsewhere (Shapiro et al., 2001), we have shown that this use of background knowledge can greatly speed the process of learning control policies. Here we focus on a different claim: that providing a learning agent with different reward signals can lead to a great variety of behaviors that still share the same overall structure. This approach to learning — which we call *programming by reward* -- should prove useful in constructing simulated agents for computer games, in supporting personalized services that must operate within certain constraints, and many other tasks.

In the following pages, we report one instance of this general framework, which we have cast in an architecture for physical agents called *Icarus*. We begin by describing the architecture's logical formalism for encoding hierarchical skills, taking examples from the task of driving an automobile. We then turn to the value functions that Icarus uses to select among applicable skills and its algorithm for using delayed rewards to update these functions. After this, we present experimental studies designed to test our hypothesis that providing such a system with different rewards can produce distinctive yet still viable policies. Finally, we examine some other approaches to learning complex skills and suggest directions for additional research on this topic.

2. The Icarus Language

Icarus is a language for specifying the behavior of artificial agents that learn. Its structure is dually motivated by the desire to build practical agent applications and the desire to support policy learning in a computationally efficient way. We responded to the first goal by providing Icarus with powerful representations. However, the desire for rapid learning suggests a simpler format that offers a clear mapping into the Markov decision process (MDP) model, since MDPs provide a conceptual framework for developing learning algorithms, and mathematical properties useful for convergence proofs. We resolved this tension by casting Icarus as a reactive computing language.

2.1 A Reactive Formalism

Reactive languages are tools for specifying highly contingent agent behavior. They supply a representation for expressing plans, together with an interpreter for evaluating plans that employs a repetitive sense-think-act loop. This repetition provides adaptive response; it lets an agent retrieve a relevant action even if the world changes from one cycle of the interpreter to the next.

Reactive languages offer a spectrum of vocabularies for expressing plans. This includes combinational logic (Agre, 1988), directed graphs (Georgeff, et al. 1985), prioritized procedures (Brooks, 1986), ordered production rules (Nilsson, 1994) and goal structures with preconditions (Schoppers, 1987). Reactive languages also support different degrees of adaptive response. Some embed reaction in an overall schema for sequential behavior, while extremely reactive languages make no commitment to control flow (because their interpreters let the world change from one state to any other recognized by the plan in exactly one time step). This format is very similar in spirit to an MDP, since both employ an iterated situation-response loop and both allow arbitrary transitions with no memory of past state.

Icarus is an instance of an extremely reactive language. It shares the logical orientation of teleoreactive trees (Nilsson, 1994) and universal plans (Schoppers, 1987), but adds vocabulary for expressing hierarchical intent, as well as tools for problem decomposition found in more general-purpose languages. For example, Icarus supports function call, Prolog-like parameter passing, pattern matching on facts, and recursion.

An Icarus program contains up to three elements: an objective, a set of requirements (or preconditions), and a set of alternate means (or methods for achieving objectives), as illustrated in Figure 1. Each of these can be instantiated by further Icarus plans, creating a logical hierarchy that terminates with calls to primitive actions or sensors. Icarus evaluates these fields in a situation-dependent order, beginning with the objective field. If the objective is already true in the world, evaluation succeeds and nothing further needs to be done. If the objective is false, the interpreter examines the requirements field to determine if the preconditions for action have been met. If so, evaluation progresses to the means field, which contains alternate methods for accomplishing the objective (primitive actions or subplans). The means field is the locus of all value-based choice in Icarus, since the objectives and requirements contain no options. In order to support this choice, the interpreter associates a value estimate with each Icarus plan. Icarus will learn to select the action or subplan that promises the largest expected reward.

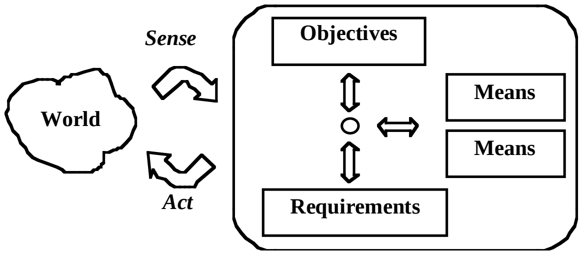

Figure 1. The structure of Icarus plans.

Icarus offers several unusual features that increase its representational power: it allows the execution of a process to be a goal, and it embeds a sequence primitive within a reactive interpreter (where reaction within a sequential plan is more common). In addition, it offers control over plan expansion; Icarus can commit to a subplan before investigating it, or it can investigate subplans and choose among the actions returned. See Shapiro (2001) for a more complete description of the language.

2.2 An Icarus Plan for Driving

Table 1 illustrates the top-level elements of an Icarus plan for freeway driving. It contains an ordered set of objectives implemented as further subplans. Icarus repetitively processes this plan, starting with its first statement every execution cycle. The interpreter employs a three-valued semantics, where every statement in the language evaluates to one of True, False, or an Action. ÔTrueÕmeans the statement was true in the world, ÔFalseÕ means the plan did not apply, and an ÔActionÕreturn identifies a piece of code for controlling actuators that addresses the objectives of the plan.

Table 1. The top level of an Icarus freeway-driving plan.

```
Drive ()
  :objective
   [ *not* (Emergency-brake())
     *not* (Avoid-trouble-ahead())
      Get-to-target-speed()
      *not* (Avoid-trouble-behind())
      Cruise()]
```

The first clause in Table 1 defines a reaction to an impending collision. If this context applies, Icarus returns the slam-on-the-brakes action for application in the world. If emergency braking is not required, evaluation proceeds to the second clause, which specifies a plan for reacting to trouble ahead, defined as a car travelling slower than the agent in the agentÕs own lane. This subplan contains options, as shown in Table 2. Here, the agent can move one lane to the left, move right, slow down, or cruise at its current speed and lane, but the plan does not include the option to speed up. Icarus makes a selection based on the long-term expected reward of each alternative.

If there is no imminent collision or trouble in front, Icarus examines the third clause of Table 1, which invokes a goal-driven subplan for bringing the agent to its target speed. This subplan causes the agent to speed up if it is traveling too slow or slow down if it is moving too fast, but it evaluates to ÒTrueÓ if the agent is currently traveling at its target speed. (Note that the fields in an Icarus plan contain default values: False for the :objective, True for :requires, and False for the :means field.)

Table 2. An Icarus plan with alternate subplans.

```
Avoid-trouble-ahead ()
  :requires
   [  bind (?c, car-ahead-center())
      velocity() > velocity(?c)
      bind (?tti,  time-to-impact())
      bind (?rd, distance-ahead())
      bind (?rt, target-speed() – velocity())
      bind (?art, abs(?rt)) ]
   :means
    [ safe-cruise(?tti, ?rd, ?art)
      safe-slow-down (?tti, ?rd, ?rt)
      move-left (?art)
      move-right (?art) ]
```

If the first three clauses in Table 1 are True, Icarus examines the fourth clause, a subplan for reacting to a faster car behind. This subplan (not shown) also contains options; it lets the agent move over or simply ignore the vehicle behind and cruise. Finally, if there is no cause to emergency brake, no trouble ahead, the agent is at its target speed, and there is no trouble behind, the fifth clause always returns an action. This causes the agent to cruise in its current lane at its current speed.

Since Icarus plans contain choice points, the interpreter needs a method of selecting the right option to pursue. In particular, we would like to know the total benefit (as opposed to the immediate return) for making a given choice on the current time step, so that the agent can maximize its prospective future reward. Icarus provides this capability by associating a value estimate with each Icarus plan. This number represents the expected future discounted reward stream for choosing a primitive action or subplan on the current execution cycle and following the policy (being learned) thereafter. Icarus computes this expected value using a linear function of current observations. For example, Avoid-trouble-ahead (Table 2) defines several parameters solely for the purpose of value estimation; the data are not required to execute any of the routines in its :means field.

The estimation architecture addresses an interesting tension in information needs. On one hand, the value of a plan clearly depends upon its context; the future of ÒdecelerateÓ is very different if the car in front is close or far. On the other hand, the cardinal rule of good programming is "hide information". We should not force Icarus programmers to define subplans with a suite of value-laden parameters that are irrelevant to performing the task at hand. Our solution is to inherit context-setting parameters down the calling tree. Thus, Avoid-trouble-ahead measures the distance to the car in front, and Icarus *implicitly* passes that parameter to the decelerate action several levels deeper in the calling tree. The programmer writes Icarus code in the usual fashion, without concern for this implicit data.

2.3 The SHARSHA Algorithm

SHARSHA is a reinforcement learning method mated to Icarus plans. It is a model-free, on-line technique that determines an optimal control policy by exploring a single, infinitely long trajectory of states and actions. SHARSHA (for State Hierarchy, Action, Reward, State Hierarchy, Action) adds hierarchical intent to the well-known SARSA algorithm (for State, Action, Reward, State, Action).

SARSA operates on state-action pairs. It learns an estimate for the value of taking a given action in a given state by sampling its future trajectory. SARSA repeats the following steps: (1) select and apply an action in the current state; (2) measure the in-period reward; (3) observe the subsequent state and commit to an action in that state; and (4) update the estimate for the starting state-action pair, using its current value, the current reward, and the estimate associated with the destination pair. In other words, SARSA bootstraps; it updates value estimates with other estimates, grounding the process in a real reward signal. Singh, et al. (in press) have shown that SARSA converges to the optimal policy and the correct values for the future discounted reward stream. The proof imposed common Markov assumptions, required an exact (tabular) representation of the true reward function, and allowed a range of action selection policies that guaranteed sufficient exploration of apparently sub-optimal choices.

SHARSHA adapts SARSA to plans with a hierarchical model of intent. In particular, it operates on stacks of state-action pairs, where each pair corresponds to an Icarus function (encoding a plan to pursue a course of action in a given situation), as depicted in Figure 2. For example, at time 1 an Icarus agent for piloting a car might accelerate to reach its target speed in order to drive, while at time 2 it might brake in order to avoid trouble as part of the same driving skill. SHARSHA employs the SARSA inner loop with slight modifications: where SARSA observes the current state, SHARSHA observes the calling hierarchy, and where SARSA updates the current state, SHARSHA updates the estimates for each function in the calling stack. The second difference is that SHARSHAÕs update operator inputs the current estimate, the reward signal, and the estimate associated with the primitive action on the next execution cycle. In principle, this primitive carries the best estimate because it utilizes the more informed picture of world state built while evaluating the Icarus program.

Our implementation of SHARSHA includes several additional features. It employs eligibility lists to speed learning, it normalizes sensor values at run-time (since the update rule can otherwise diverge), it supports multiple exploration policies, and it employs linear value approximation functions in place of tabular forms. SHARSHA learns the coefficients of these linear mappings from delayed reward. We have proven SHARSHAÕ converge properties under a common set of Markov assumptions (Shapiro, 2001).

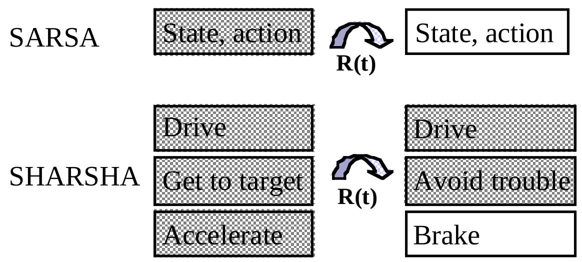

Figure 2. A comparison of SARSA and SHARSHA.

3. An Experiment with Programming by Reward

Now that we have introduced the Icarus architecture, we turn to an experiment on its advantages for agent design. In particular, we would like to know if we can employ programming by reward to build interesting agents. Here, an engineer implements a base of skills that contain options, and does this once per application domain, while users create individual agents by defining distinct reward functions that serve as the target of learning.

The key question is whether shared skills possess enough flexibility to support this programming model. We believe the answer is yes, and we codify that conjecture in the *separation hypothesis*: that the behavioral differences among individuals performing common physical tasks are due to the action of distinct preferences over the same set of skills. For example, we all know how to drive, but some of us are passive and others more aggressive drivers. If the hypothesis holds, skills and preferences are loosely coupled and we can develop agents via programming by reward. If it is false, skills and reward are tightly coupled, and a change to one requires a change in the other. Any method that considers them separable is doomed to failure.

We investigate the separation hypothesis through a simple qualitative experiment. We define a set of reward functions, use them to train the Icarus skill outlined in Tables 1 and 2, and then compute and compare various behavioral measures. We begin by discussing agent-held reward functions.

3.1 Agent-held Reward Functions

In order to test the model of programming by reward, we defined a set of qualitatively different reward functions. All of them are linear in their feature values, and Table 3 associates their features with mnemonic names. The airport driver is solely motivated by the desire to get to the airport on time. It becomes less happy as its velocity deviates from target speed. The safe driver wants to avoid collisions. Its reward function penalizes small times to impact with cars in front and cars behind. The shorter the time to impact, the larger the penalty, with times greater than 100 seconds having no reward. The goldfish driver has an imaginary fishbowl as luggage, and does not want maneuvers to upend the fish. Alternatively, we can think of the goldfish driver as a bit queasy; its reward function penalizes all forms of maneuver. The reckless teenager is out for thrills; it garners reward for near misses, and also cares about maintaining its cruising speed. The crowd lover and the crowd hater desire the expected things; their reward increases (or decreases) with the number of surrounding cars. Note that the rewards are calculated once every execution cycle, and the learning system seeks to acquire the greatest reward stream over time.

Table 3. Definitions for agent-held reward functions.

	Time to impact ahead	Time to impact behind	Deviation from target speed	Slowing down	Speeding up	Changing lanes	Nearby Cars
Airport driver			–				
Safe driver	+	+					
Goldfish driver				–	–	–	
Reckless teenager	–	–	–				
Crowd Lover							+
Crowd Hater							–

3.2 A Profile of Learned Behavior

We use each of the above reward functions to develop agent personalities by employing them as the target of policy learning. We conduct ten 32,000-iteration training runs for each driver, and the following figures discuss averages computed over the final 20,000 iterations of each run. In all cases, we initialize the driverÕ velocity to a random number between zero and its target speed (62 mph), and all of its value-estimation functions to zero. Figure 3 focuses on behavioral measures, using the safe driverÕ score as the unit quantity. We analyze the maximum and minimum values in each category.

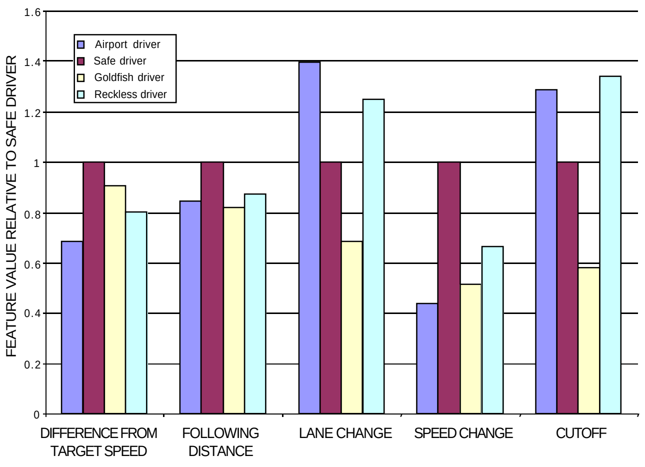

Figure 3. A profile of learned behaviors in a low-density traffic environment.

The first measure is the absolute value of the agentÕ difference from its target speed. The fact that the airport driver has the lowest score is not surprising, since its reward function directly penalizes non-zero values. However, the safe driver shows the highest difference from target speed in the chart. It is not motivated (positively or negatively) by this quantity, but apparently, it readily adjusts its velocity to avoid potential collisions (i.e., short times to impact).

The safe driver also shows the largest following distance. This makes intuitive sense, since safe drivers know that tailgating produces potential collisions. (If the car in front slows down, the safe driver inherits a significant penalty.) In contrast, the goldfish driver has the shortest following distance (although it is close to the airport driverÕ). We explain this observation by a cruise control effect: drivers who resist velocity changes will tend to creep up on the car in front. Both the airport and goldfish drivers contain this bias. Note that none of these agents assign direct value to following distance in their reward functions.

The airport driver displays the highest number of lane changes. This makes sense if it is maneuvering to maintain its target speed. The goldfish driver shows the fewest, as it is centrally motivated not to make such changes. The speed change results are similar: the airport driver is directly biased against deviating from target speed, while the safe driver freely adjusts its speed to avoid potential impacts.

Finally, the goldfish driver performs the fewest cutoff actions (defined as a lane change in front of a faster vehicle), as it is motivated to avoid all maneuvers. In contrast, the reckless driver actively seeks potential collisions, as they contribute positive terms to its reward.

Note that the driving program prevents the reckless teenager from simply colliding with the car in front; its opportunity to learn is confined to an allowable realm. Said differently, the agentÕ skills ensure reasonable behavior. Its reward function is irrelevant whenever the behavior is determined, and relevant only when choice is allowed. This design frees us to construct reward functions in an unconstrained way.

3.3 Learned Lane Preferences

Figure 4 illustrates an emergent property of programming by reward. We plot the agentÕ occupancy in the different freeway lanes, and note that the crowd lover evolves a slight preference for the middle lane, while the crowd hater generates a strong preference for the right hand lane. These preferences were never encoded in the reward functions, although the results make intuitive sense. A car in the center lane has the potential to encounter up to six adjacent vehicles (good for a crowd lover), while a car in the right or left lane can have a maximum of four neighbors. While it is clear that the crowd-hater would avoid the center lane, it is unclear why it preferred the right lane to the left. The (fixed) control policies of the cars constituting the freeway traffic do act to sort vehicles into lanes by speed. Perhaps there is a smaller difference between the average speed in the right and center lanes than between the left and center lanes. If so, the crowd hater will encounter fewer cars per unit time if it gravitates to the right.

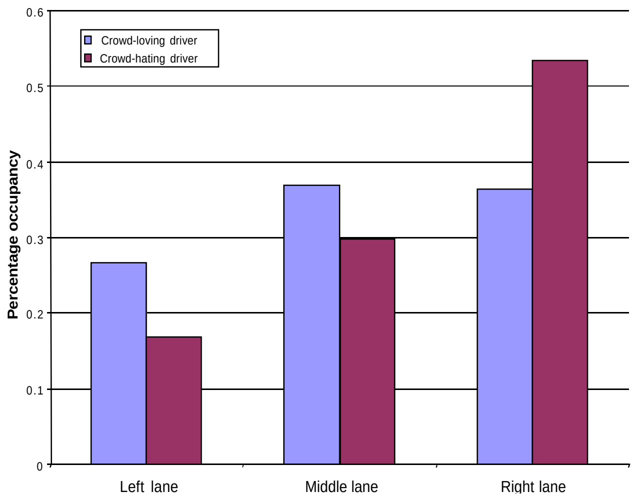

Figure 4. Lane preferences learned by the crowd-loving and crowd-hating drivers.

3.4 Driver Behavior Across Two Domains

It is clear that we can generate distinct behavior via programming by reward, but we would also like to know if that behavior is in some sense robust to environmental change. We investigate this question by training the same agents in a high, vs a low density traffic scenaior. Figure

5 provides the results. Here, we take the performance of the safe driver in low-density traffic as the unit quantity so that we can compare behavior both within and across domains.

Our first observation is that the absolute magnitudes of the metrics differ as we move between scenarios. It is generally harder to maintain target speed in high-density traffic, following distances shrink, it becomes more difficult to change lanes, and the agents have to adjust their speed more often in order to respond to other traffic. These changes make sense, as they are largely forced upon the agents by increased traffic density.

A more striking observation is that the behavioral profiles are beautifully preserved across domains. Although the number of instances of any given behavior changes, the shapes of the curves are virtually identical in low and high-density traffic. There are only two shifts in relative order, for the maximum number of lane changes and minimum number of speed changes.

This constancy of behavior provides evidence that programming by reward shapes agent behavior in a predictable way, and that it can be used in a development model where agents are trained in a test domain, and deployed in an application environment.

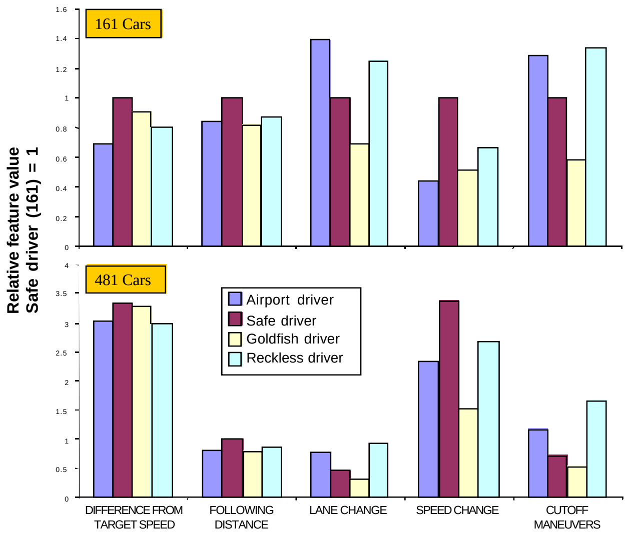

Figure 5. A comparison of learned behavior in two domains.

3.5 Searching the Space of Reward Functions

Now that we have examined the relation between a reward function and the behavior it generates, it is worth asking the question in the opposite direction. Can we generate a specific, predefined behavior via programming by reward?

We pursued this question by attempting to duplicate (in a qualitative sense) the behavior of one of the authors who is a particularly aggressive driver. We did this by searching across the space of possible reward functions, seeking to minimize the agentÕs following distance while simultaneously maximizing the number of cutoff maneuvers.

The results were both positive and negative. On the positive side, we succeeded in generating a 10-fold increase in the number of cutoff maneuvers performed by the Ôroad rage driverÕ relative to the reckless teenager, as shown in Figure 6. However, we were only able to do so by introducing a slight modification to the shared driving skill; we gave both drivers the option to change lanes in the absence of a slower car in front, or a faster car behind. The original skill lacked the flexibility to support the desired (extremist) behavior.

This experiment also generated an interesting strategic lesson for programming by reward. We discovered that it was far more successful to penalize the roadrage driver as it was being passed by other cars, rather than to reward it when it cut off other vehicles. The reason is that it there are more opportunities to learn from persistent conditions than momentary events.

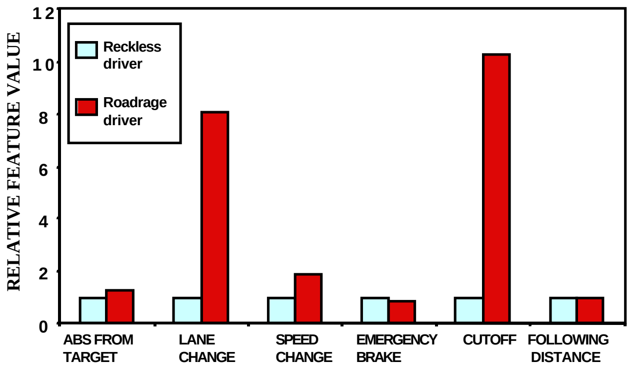

Figure 6. Using a reward function to generate road rage.

4. Related Work on Control Learning

Earlier we contrasted our framework for control learning with other approaches, but the previous work on this topic and its differences from our own deserves a more detailed discussion. Here we consider four distinct paradigms for learning control policies from experience that have appeared in the literature.

One body of research focuses on architectures for intelligent agents, with two well-known examples being Soar (Laird & Rosenbloom, 1990) and Prodigy (Minton, 1990). These systems represent knowledge about legal actions as production rules or logical operators, which they utilize during problem solving and execution. Because this knowledge predicts the effects of operators, they can learn from the results of problem solving, rather than relying, as does Icarus, on feedback from the environment. Both architectures learn preferences over actions, states, and goals, but they encode these as logical rules, in contrast with Icarus' use of utility functions. The ACT-R architecture (Anderson 1993) associates strengths with learned production rules based on their success in achieving goals, but these specify a scalar value rather than a numeric function of environmental features.

An alternate framework is to learn control policies from observations of another agent's behavior by transforming traces into supervised training cases. Such *behavioral cloning* typically generates knowledge in the form of decision trees or logical rules (e.g., Sammut, 1996; Urbancic & Bratko, 1994), though other encodings are possible (Anderson, Draper, & Peterson, 2000). Unlike Icarus, these systems typically acquire control knowledge from scratch, but one could utilize high-level plans to parse a behavioral trace and thus constrain the cloning process. Similarly, our framework could be extended to learn from observational traces, using them to update the utility function associated with each skill, much as some adaptive interfaces (e.g., Rogers, Fiechter, & Langley, 1999) induce such functions from user choices.

A larger body of research emphasizes learning policies from delayed external rewards. Within this framework, some methods represent their control knowledge as logical rules that state the conditions under which particular actions are desirable (e.g., Grefenstette, Ramsey, & Schultz, 1990). Others achieve the same effect with different formalisms like multilayer neural networks (e.g., Moriarty & Langley, 1998). In this paradigm, learning involves a search through the space of policies, using genetic or other methods, guided by the rewards that alternative candidates receive from the environment. The search process typically starts from scratch, but, clearly, it could be aided by starting from skills that specify legal actions. However, this framework does not lend itself to a division between legal skills and preferences stated as utility functions.

An alternative approach to learning from delayed rewards encodes policies as utility functions, an idea that plays a central role in Icarus. These functions are typically stored in a table that associates an estimated value with each state-action pair, but some work instead uses approximations. This mapping is learned through methods like Q learning (Watkins & Dayan, 1992), which update the estimated value of a state-action pair based on the discounted reward that it produces. Most research on estimating value functions in this manner emphasize learning from scratch, though some work on hierarchical reinforcement learning (e.g., Andre & Russell, 2000; Dietterich, 2000; Parr & Russell, 1998; Sutton et al., 1998) provides the learner with background knowledge. Our approach fits most comfortably within this framework, but Icarus' role as an architecture that supports programming by distinguishes it from other research along these lines.

In summary, our approach to representing, using, and learning control policies has many common features with other work on this topic. However, Icarus differs from previous systems in its clear separation of control knowledge into logical skills and numeric utility functions, which we claim supports considerable variety in agent behavior while keeping it within domain constraints. This division in turn lets us program agents by reward to exhibit quite different behaviors.

5. Conclusions

Our experiments have shown that we can produce qualitatively distinct agents via programming by reward. That is, we can construct one set of skills, define individual agents by encoding suitable reward functions, and train those agents by letting them learn from experience. The reward functions are easy to construct and their content is unconstrained.

These results provide evidence in support of the separation hypothesis. If it holds more generally than in our traffic domain, the roles of preferences and skills may be sufficiently decoupled to support programming by reward in practical applications. If so, we will be able to create entire families of agents in real applications without writing new skills. This is important because skill development is time consuming and expensive, hard work.

While this paper emphasized the use of reward functions in a programming metaphor, we also designed a reward function to accomplish a specific objective. This required a search process, but we can define a more direct method. In particular, we have shown elsewhere (Shapiro, 2002) that it is always possible to align an agentÕs reward function with human utility, such that the agent will do the best possible job it can for that person as a byproduct of learning to maximize its own reward. This is an open area for future research.

In summary, the Icarus architecture and the methodology of programming by reward appear to provide an efficient means of encoding desired behavior. The approach merits an in-depth look in a variety of application domains, e.g., for constructing conversational agents, non-player characters in computer games, and household robots whose personalities are tailored to their owners.

References

Agre, P. (1988). *The dynamic structure of everyday life.* Tech Report AI-TR-1085, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Anderson, C., Draper, B., & Peterson, D. (2000). Behavioral cloning of student pilots with modular neural networks. *Proceedings of the Seventeenth*

International Conference on Machine Learning (pp. 25-32). Stanford: Morgan Kaufmann.

Anderson, J. R. (1993). *Rules of the mind.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Andre, D., & Russell, S. J., 2001. (NIPS-2000) Programmable reinforcement learning agents. *Proceedings of the 13th Conference on Neural Information Processing Systems*. MIT Press, pages 1019-1025.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation, 2, 1*.

Dietterich, T.G. (2000). State abstraction in MAXQ hierarchical reinforcement learning. *Advances in Neural Information Processing Systems, 12*. MIT Press.

Georgeff, M., Lansky, A., & Bessiere, P. (1985). A procedural logic. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann.

Grefenstette, J. J., Ramsey, C. L., & Schultz, A.C. (1990). Learning sequential decision rules using simulation models and competition. *Machine Learning*, *5*, 355--381.

Laird, J. E., & Rosenbloom, P. S. (1990). Integrating execution, planning, and learning in soar for external environments. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp.1022--1029). Boston, MA: AAAI Press.

Minton, S. N. (1990). Quantitative results concerning the utility of explanation-based learning. Artificial Intelligence, *42*, 363--391.

Moriarty, D., & Langley, P. (1998). Learning cooperative lane selection strategies for highways. *Proceedings of the Fifteenth National Conference on Artificial Intelligence* (pp. 684--691). Madison, WI: AAAI Press.

Nilsson, N. (1994). Teleoreactive programs for agent control. *Journal of Artificial Intelligence Research*, *1*, 139-158.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems, 10* (pp. 1043-1049). MIT Press.

Rogers, S., Fiechter, C., & Langley, P. (1999). An adaptive interactive agent for route advice. *Proceedings of the Third International Conference on Autonomous Agents* (pp. 198--205). Seattle: ACM Press.

Sammut, C. (1996). Automatic construction of reactive control systems using symbolic machine learning. *Knowledge Engineering Review*, *11*, 27--42.

Schoppers, M. (1987). Universal Plans for reactive robots in unpredictable environments. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 1039-1046). Morgan Kaufmann.

Shapiro, D. (2002). User-agent value alignment. *Stanford Spring Symposium, Workshop on Safe Learning Agents*. Stanford, CA.

Shapiro, D. (2001). *Value-driven agents*. PhD thesis, Department of Management Science and Engineering, Stanford University, Stanford, CA.

Shapiro, D., Langley, P., & Shachter, R. (2001). Using background knowledge to speed reinforcement learning in physical agents. *Proceedings of the Fifth International Conference on Autonomous Agents* (pp. 254--261). Montreal: ACM Press.

Singh, S., Jaakola, T., Littman, M., & Szepesvari, C. (in press). Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning.*

Sutton, R. S., Precup, D., & Singh, S. (1998). Intra-option learning about temporally abstract actions. *Proceedings of the Fifteenth International Conference on Machine Learning* (pp. 556-564). Morgan Kaufmann.

Urbancic, T., & Bratko, I. (1994). Reconstructing human skill with machine learning. *Proceedings of the Eleventh European Conference on Artificial Intelligence* (pp. 498--502). Amsterdam: John Wiley.

Watkins, C., & Dayan, P. (1992). Q-learning. *Machine Learning, 8,* 279-292.