Cumulative Learning of Hierarchical Skills

Pat Langley
Seth Rogers
Computational Learning Laboratory
Center for the Study of Language & Information
Stanford University, Stanford, CA 94305 USA

Abstract

In this paper, we review Icarus, *a cognitive architecture that utilizes hierarchical skills and concepts for reactive execution in physical environments. In addition, we present two extensions to the framework. The first involves the incorporation of means-ends analysis, which lets the system compose known skills to solve novel problems. The second involves the storage of new skills and concepts that are based on successful means-ends traces. We report experimental studies of this mechanism in the blocks world, which show that learning operates in a cumulative manner and that it reduces the time required to handle new tasks. We conclude with a discussion of related work on learning and directions for future research.*

1. Introduction and Motivation

Research on cognitive architectures (Newell, 1990) attempts to understand the computational infrastructures that support intelligent behavior. A specific architecture specifies the aspects of a cognitive agent that remain the same across time and over different domains, and typically makes strong commitments about the representation of knowledge structures and the processes that operate them. Learning has been a central concern in most architectural research, with a variety of mechanisms having been proposed to model the acquisition of knowledge from experience.

In this paper we review Icarus, a candidate architecture that diverges from its predecessors on a number of dimensions. One key difference is that most frameworks focus on production systems, which encode knowledge as a 'flat' set of condition-action rules, whereas Icarus provides explicit support for hierarchies of both concepts and skills. In addition, most cognitive architectures evolved from theories of human problem solving, and thus focus on mental phenomena. In contrast, Icarus is mainly an execution architecture that perceives and reacts to external environments.

However, Icarus' commitment to hierarchical structures raises important questions about their origin. Also, the architecture's emphasis on execution does not mean that mental activities such as problem solving are unimportant, since they can let an agent handle novel tasks for which stored knowledge is unavailable. The central hypothesis of this paper is that hierarchical skills and concepts arise, at least in many cases, from problem-solving behavior, and that, once learned, these structures can be used to support reactive execution in the environment.

In the sections that follow, we review Icarus' representation and organization of concepts and skills, along with the categorization and execution processes that utilize them. After this, we present a new module that interleaves means-ends problem solving with execution when known skills are insufficient to solve a task. Next we describe a mechanism for creating generalized skills and concepts from traces of successful problem solving that supports both incremental and cumulative learning. We then report experiments with this learning mechanism that demonstrate its ability to reduce effort on new problems and that examine effects of training order. In closing, we discuss earlier research on learning problem-solving knowledge and cumulative learning, along with some directions for future work.

2. Representation and Organization

Like other cognitive architectures, Icarus makes commitments to its representation of knowledge, the manner in which it is organized, and the memories in which it resides. Here we discuss the framework's long-term and short-term memories, including the formalism used to encode their contents. We will take our examples from the blocks world, since many readers will find this domain familiar.

One of ICARUS' long-term memories stores Boolean concepts that describe situations in the environment. These may involve isolated objects, such as an individual block, but they can also characterize physical relations among objects, such as the relative position of two blocks. Long-term conceptual memory contains the definitions of these logical categories. Each element specifies the concept's name and arguments, along with four optional fields – `:percepts`, which describes perceptual entities that must be present; `:positives`, which indicates lower-level concepts that must match; `:negatives`, which specifies lower-level concepts that must not match; and `:tests`, which specifies numeric relations that must be satisfied.

Table 1 presents some concepts from the blocks world. For example, *on* describes a situation in which ICARUS perceives two blocks with the same x position and the bottom of one has the same y position as the top of the other. The concept *clear* refers to a single block, but one that cannot hold the relation *on* to any other, as specified in its `:negatives` field.[1]

Definitions of this sort organize ICARUS categories into a conceptual hierarchy. Some concepts may be defined entirely in terms of perceptual conditions and numeric tests, but many incorporate other concepts in their definitions. This imposes a lattice structure on the memory, with more basic concepts at the bottom and more complex concepts at higher levels. The resulting hierarchy is similar in spirit to early models of human memory like EPAM (Feigenbaum, 1963), as well as to frameworks like description logics.

ICARUS also incorporates a second long-term memory that stores knowledge about skills it can execute in the environment, including their conditions for application and their expected effects. Each skill has a name, arguments, and a set of optional fields. The `:start` field specifies the concepts that must hold to initiate the skill, whereas the `:requires` field indicates conditions that must hold throughout the skill's execution, which may require multiple cycles to complete. The `:effects` field specifies a conjunction of concepts that, taken together, describe the situation the skill will produce when done. For example, Table 2 shows the skill *pickup*, which refers to two blocks. These must satisfy the single start condition, *(pickupable ?block ?from)*, which (as shown in Table 1) requires *(on ?block ?from)*, *(clear ?block)*, and *(hand-empty)*. The skill's only stated effect is to make *(holding ?block)* true.

Each ICARUS skill also includes a field that specifies how to decompose that skill. Most examples in the table utilize the `:actions` field, which refers to opaque actions the agent can execute directly in the environment. For instance, *unstack* invokes both **grasp*, which grasps a block, and **vertical-move*, which moves the hand in the vertical direction. However, nonprimitive skills like *pickup-puton* instead include the `:ordered` field, which specifies one or more subskills of which it is composed, in this case the primitive skills *pickup* and *puton*.[2]

Table 1. Some ICARUS concepts for the blocks world, with variables indicated by question marks.

```
(on (?block1 ?block2)
 :percepts  ((block ?block1 xpos ?x1 ypos ?y1)
             (block ?block2 xpos ?x2 ypos ?y2
                    height ?h2))
 :negatives ((clear ?block2))
 :tests     ((equal ?x1 ?x2) (>= ?y1 ?y2)
             (<= ?y1 (+ ?y2 ?h2))))
(clear (?block)
 :percepts  ((block ?block))
 :negatives ((on ?other ?block)))
(pickupable (?block ?from)
 :percepts  ((block ?block)(table ?from))
 :positives ((ontable ?block ?from)
             (clear ?block)
             (hand-empty)))
(pickup-putonable (?block ?from ?to)
 :percepts  ((block ?block)(table ?from)(block ?to))
 :positives ((pickupable ?block ?from)
             (clear ?to)))
```

In fact, ICARUS lets one specify multiple ways to decompose a given concept or skill, much as a Prolog program can include more than one Horn clause with the same head. In addition, each skill decomposition can include a value function that encodes the utility expected if the agent executes the skill with that decomposition. Neither capability plays an important role in this paper, but we have described them in some detail elsewhere (Choi et al., in press).

In addition to long-term memories, which encode stable knowledge about a domain, ICARUS includes short-term stores that can change more rapidly. These make contact with long-term concepts and skills, but they represent temporary beliefs about the environment and intended activities. In particular, the *perceptual buffer* contains descriptions of physical entities that correspond to the output of sensors. For the blocks world, this includes literals like *(block B xpos*

[1] Note that on also mentions clear in its :negatives field, giving ICARUS some ability to state mutually exclusive concepts.

[2] ICARUS also supports an unordered field for subskills that can be executed in any order, but they play no role in this paper.

Table 2. Some ICARUS skills for the blocks world.

```
(pickup (?block ?from)
 :percepts ((block ?block)
            (table ?from height ?h))
 :start    ((pickupable ?block ?from))
 :actions  ((*grasp ?block)
            (*vertical-move ?block (+ ?h 10)))
 :effects  ((holding ?block)))
(puton (?block ?to)
 :percepts ((block ?block)
            (block ?to xpos ?x ypos ?y height ?h))
 :start    ((putonable ?block ?to))
 :actions  ((*horizontal-move ?block ?x)
            (*vertical-move ?block (+ ?y ?h))
            (*ungrasp ?block))
 :effects  ((on ?block ?to)(hand-empty)))
(pickup-puton (?block ?from ?to)
 :percepts ((block ?block)(block ?from)(table ?to))
 :start    ((pickup-putonable ?block ?from ?to))
 :ordered  ((pickup ?block ?from)
            (puton ?block ?to))
 :effects  ((on ?block ?to)))
```

10 ypos 2 width 2 height 2), which specify the position and size of individual blocks. Moreover, the *short-term conceptual memory* contains beliefs about the environment that the agent can infer from those present in its perceptual buffer and its long-term concept memory. For instance, this might contain the instance *(on B C)*, which is an instance of the *on* concept in Table 1. Finally, ICARUS includes a *short-term skill memory* that contains the agent's intentions about skill instances it plans to execute, which lets the system engage in behavior that persists over time. Each literal specifies the skill's name and its arguments, as in *(puton B C)*.

3. Categorization and Execution

Like most cognitive architectures, ICARUS operates in cycles. On each iteration, the system updates its perceptual buffer by applying sensors to every object within a given distance. This produces perceptual elements that initiate matching against long-term concepts. The matcher checks to see which concepts are satisfied, adds each matched instance to conceptual short-term memory, and repeats the process on the expanded set. In this way, ICARUS infers all instances of concepts that are implied by its conceptual definitions and the contents of the perceptual buffer. In the blocks world, the agent would first update its descriptions of the blocks and the table, then infer concepts like *on*, and then infer concepts like *unstackable*.

On each cycle, the architecture examines the intentions in short-term skill memory to determine which, if any, apply to the current situation and which one has the highest expected value. For each skill instance, ICARUS accesses all expansions of the general skill to see if they are applicable. A skill is applicable if, for its current variable bindings, its `:effects` field does not match, the `:requires` field matches, and, if the system has not yet started executing it, the `:start` field matches the current situation. Also, for higher-level skills, at least one subskill must be applicable. Because this test is recursive, a skill is applicable only when ICARUS can find at least one acceptable path downward to an executable action. ICARUS considers all acceptable paths downward through the skill hierarchy, returning the path with the highest expected value.

When expected values are equal, ICARUS selects one of the skill paths at random, as we assume in this paper. For example, suppose the agent has the intention *(pickup-puton A table B)* in a situation where the concept instances *(pickup-putonable A table B)* and *(pickupable A table)* hold. This means the path *((pickup-puton A table B), (pickup A table))* is acceptable and would be considered for execution. If selected, the *pickup* skill would alter the environment, making acceptable the path *((pickup-puton A table B), (puton A B))* on the next cycle. This would produce a state that satisfies the effects of *(pickup-puton A table B)*, making any path in which it occurs unacceptable.

The architecture handles a skill differently depending on how it is decomposed. For primitive skills that include an `:actions` field, ICARUS executes each of the physical actions, one after another, on a single cycle. For higher-level skills that have an `:ordered` field, it treats the list as a reactive program that considers each subskill in reverse order. If the final subskill is applicable, then it considers only paths that include that subskill. Otherwise, it considers the penultimate skill, the one before that, and so forth. Presumably, the subskills are ordered because later ones are closer to the parent skill's objective, and should be preferred over earlier ones when applicable.

4. Means-Ends Problem Solving

As explained above, the initial version of ICARUS can execute complex hierarchical skills in a reactive manner, but it assumes that these skills are already present in long-term memory. Although much human behavior involves the application of such routine skills, people also have the ability to solve novel problems that require the combination of existing knowledge elements.

To model this capability in ICARUS, we have introduced a variant of means-ends analysis (Newell, Shaw,

& Simon, 1960) that operates over the architecture's knowledge structures. Traditional means-ends problem solving selects some unsatisfied aspect of the goal state to achieve, then selects an operator that would achieve it. If that operator's preconditions match the current state, it is applied; otherwise, the method selects an unsatisfied precondition to achieve, selects an operator that would achieve it, and so on. Once a condition is achieved, the process is repeated until the original goal description is satisfied. This may require search, which is typically pursued in a depth-first manner. Means-ends analysis has been implicated strongly in human problem solving, and it forms the central technique in the PRODIGY architecture (Minton et al., 1989).

ICARUS implements a variant of this mechanism with a goal stack that contains goal elements in an ordered list. Each goal element specifies an objective (a desired goal literal) and whether it involves backward chaining off a concept definition or a skill. If the latter, then the element may specify a skill that achieves the objective. Also, a goal element may have a 'failed' field for skills or concepts that it has tried and rejected. On each cycle, ICARUS takes one of six steps:

- If the top entry E on the stack has an associated skill instance S that is allowable, in the sense described above, then ICARUS selects a skill path for S and executes it in the environment.
- If the top entry E on the stack has an associated skill instance S that is not executable, then ICARUS adds a new entry on top of the stack with the start condition of skill S as its objective.
- If the top entry E on the stack has objective O but does not have an associated skill, ICARUS retrieves skills that include O in their effects field, selects an instance S from this set, and associates S with E.
- If ICARUS retrieves no skills that would achieve objective O, it determines which instantiated subconcepts of O are not met, selects one (C) at random, and adds a goal element with C as its objective.
- If the objective O for the top entry E on the stack is satisfied by the current environmental state, then ICARUS pops E from the stack.
- Otherwise, if the system cannot find a skill instance that does not appear in the entry E's failed field, and if chaining off the unmatched elements of E's objective O has failed, then it pops E from the stack and stores O in the failed field of E's parent.

Each of these activities takes a single cycle of the architecture. Because reasoning about how to achieve an objective can require many manipulations of the goal stack, it takes more cycles than executing a hierarchical skill for the same objective, even when the agent does not have to backtrack.

Search enters into this formulation in two places. One involves backward chaining off the unmatched elements of a concept definition. Here ICARUS selects a literal randomly from those not yet tried and keeps track of literals it has failed to achieve. The other involves backward chaining off skills that, if executed, would achieve the objective of the current stack entry. Here ICARUS prefers skill instances that have not yet failed and that have the fewest expanded `:start` conditions unmet by the current environmental state, with fully matched conditions being the most desirable. If two or more candidates tie on this criterion, it prefers skill instances that have a shorter expected duration, and if ties remain, it selects a candidate at random.

Taken together, these biases produce a heuristic version of means-ends analysis. However, this problem-solving method is tightly integrated with the execution process. ICARUS backward chains off concept or skill definitions when necessary, but it executes the skill associated with the top stack entry as soon as it becomes applicable. Moreover, because the architecture can chain over hierarchical reactive skills, their execution may continue for many cycles before problem solving is resumed. In contrast, most models of human problem solving and most AI planning systems focus on the generation or the execution of plans, rather than interleaving the two processes.

Of course, executing a component skill before constructing a complete plan can lead an agent into difficulties, since it is harder to backtrack in the world than in one's head. This strategy may well lead to nonoptimal behaviors, but human intelligence is more about satisficing than optimizing, and interleaving problem solving with executing requires far less memory than constructing a full plan before executing it. However, it can produce situations from which the agent cannot recover. Thus, if ICARUS has not achieved the top-level objective in a goal stack within N cycles, it resets the environment in the original situation and tries again, with no memory of its previous attempts.

5. Learning from Problem Solving

In the previous pages, we described two facets of ICARUS: its ability to execute hierarchical skills on familiar tasks and its ability to use problem solving to handle novel tasks. The first lets the system operate efficiently, but skills are tedious to construct manually, whereas the second gives the system flexibility but requires reasoning and means-ends search. We believe that humans also have both capabilities, but that they also use learning to transform the results of successful problem solving into hierarchical skills. We would like to incorporate a similar capability into ICARUS.

However, we want our learning mechanisms to satisfy certain constraints that appear to hold for human skill acquisition. One is that learning should take advantage of existing knowledge, such as the definitions of current skills and concepts. Another characteristic is that learning should be incremental and interleaved with the problem-solving process. Taken together, these imply that learning should be *cumulative* in the sense that it builds directly on the results of previous learning. The literature on computational learning contains remarkably few examples of such cumulative acquisition of knowledge.

Our extension of ICARUS achieves this effect through a form of impasse-driven learning that is tied closely to its problem-solving and execution processes. As in SOAR (Laird et al., 1986), the purpose of skill learning is to avoid such impasses in the future. Thus, whenever the architecture achieves an objective that is associated with an entry in the goal stack, this provides an opportunity for learning. The system acquires three distinct forms of skill, which we describe in turn.

The first type of skill results from situations in which ICARUS has attempted to execute a skill instance S to achieve an objective O, but found its start conditions unsatisfied and selected another skill instance, P, to achieve them. Once both skills have been executed successfully and the objective reached, the system constructs a new skill N that has P and S as ordered subskills. The objective of N is the original objective, O, and the start condition is a new concept, C, that includes the conditions of O that were satisfied initially, the preconditions of S that were satisfied initially, and the start conditions of P. The definitions have their arguments replaced by variables in a consistent manner. For example, the skill *pickup-puton* in Table 1 might be learned from a problem that involved executing *(pickup A table)* followed by *(puton A B)* to achieve the goal *(on A B)*.

The other types of skills result from situations in which the problem solver could not find a skill to achieve an objective O, and thus created as subgoals the literals $\{O_1, O_2, \ldots, O_n\}$ from the unsatisfied conditions of O's conceptual definition. Suppose these subgoals have each been achieved in turn by executing the skill instances $\{S_1, S_2, \ldots, S_n\}$, respectively, thus satisying the parent goal O. When this occurs, ICARUS constructs a new skill N with ordered subskills $\{S_1, G_2, \ldots, G_n\}$. Each G_k is a "guard" skill with S_k as a single subskill, with no effects, and with $\{O_1, \ldots, O_{k-1}\}$ as its start conditions, which ensure that S_k is invoked only after these objectives have been met. Their parent skill N has as its effect O and has, as its start condition, a new concept C that includes both the elements of O and the elements of S_1, …, and S_n that were satisfied initially. Again, specific arguments are replaced by variables in a consistent way.

We have emphasized the construction of hierarchical skills, but, as noted above, ICARUS also acquires new concepts in the process. These play the role of start conditions for the new skills and ensure they are executed only when appropriate. Thus, one can view these concepts as functionally motivated, even though their definitions are purely structural. For example, the concept *(pickup-putonable ?block ?from ?to)* created as the start condition of skill *pickup-puton* above is defined as the conjunction of *(pickupable ?block ?from)* and *(clear ?to)*, which is the situation in which executing *(pickup ?block ?from)* followed by *(puton ?block ?to)* will achieve the effect *(on ?block ?to)*.

These learning mechanisms are fully incremental, in that each learning event draws on a single problem-solving experience and thus requires no memory of previous ones. They even support within-trial learning, since skills acquired on one subproblem may be used to handle later subproblems. The learning processes also build on existing knowledge, since the construction of new skills and concepts involves the composition of those used in a training problem's solution. Taken together, these support a form of cumulative learning, in which ICARUS learns skills and concepts on one problem, uses them in solving a later problem, and incorporates them into still higher-level skills and concepts.

6. Experiments with Hierarchy Learning

Initial studies with the blocks world confirmed that the extended version of ICARUS learns hierarchical skills and concepts in the manner described. Moreover, they revealed that, when given the same task to solve a second time, the system utilizes this knowledge to handle it without problem solving, although this does not mean it completes the problem in a single cycle. Recall that, unlike traditional cognitive architectures, ICARUS resorts to problem solving only to enable execution, and it must still execute its acquired skills to reach an objective. Thus, for a problem that requires four primitive steps, the system takes six cycles the second time it encounters this task, with one to retrieve the hierarchical skill and one to realize that it has finished.

For this domain, ICARUS learns skills for achieving particular configurations of blocks from particular initial configurations, along with concepts for the start conditions of each skill and subskill. Yet because the system generalizes its learned structures beyond the specific instances on which they are based, it can handle without problem solving any task that is isomorphic to one it has already solved. This isomorphism must involve the same objective and have the same subconcepts satisfied or unsatisfied in the initial environment.

However, we desired more than anecdotal demonstrations that the new mechanisms supported cumulative learning of skills and concepts. We also wanted evidence from systematic experiments that this learned knowledge produces more effective behavior. To this end, we examined the state space for blocks world problems that involve three blocks. If one ignores isomorphisms, then there are five problems that can be solved in two primitive steps, eight tasks solvable in four steps, nine six-step problems, and four eight-step problems.[3] These 26 tasks constituted both the training and test problems for the study.

We provided the system with four primitive skills and ten concepts, including one for the desired state, that were sufficient, in principle, to solve these blocks-world problems. We then presented it with these problems in sequence, using each task as a training problem but also recording the number of cycles required to complete it. Because misguided search combined with execution can lead the problem solver into undesirable physical states, we told it to halt if it had not finished a run within 50 cycles and start over from the initial state. However, it could attempt a given problem only ten times, and thus spend at most 500 cycles before giving up entirely. We also limited the stack depth to six goal elements. We enforced these constraints for reasons of practicality and because we think they reflect the manner in which humans tackle novel problems.

We ran Icarus on the 26 blocks-world problems, ordering them by difficulty class (two-step tasks first and eight-step tasks last) but randomly within a class. The intuition was that the system would learn more effectively if we presented it first with simpler problems, which it could then use in solving more difficult ones. To this end, Icarus retained skills and concepts acquired on successful runs for use in later tasks. However, if the system failed on a given run, it removed any skills and concepts created during that run, to prevent them influencing behavior on later attempts. We ran Icarus over 200 randomly generated problem orders and averaged the number of cycles required at each level of experience. As a control, we also ran the architecture with its learning mechanisms off on another 200 random problem sets.

Figure 2 shows the result of this experiment, including 95 percent confidence intervals around each mean. The two curves show clearly that learning reduces the total cycles required to solve problems in the blocks world. Both curves are step functions that increase with problem difficulty, as one would expect. Remember that none of the problems are isomorphic, although they may involve isomorphic subtasks. The results suggest that Icarus takes advantage of that similar substructure to reduce its effort on later problems. In a typical run on 26 problems, the system constructed 54 new concepts and 79 skills, including 14 skill-chaining skills, 36 concept-chaining skills, and 29 guard skills.

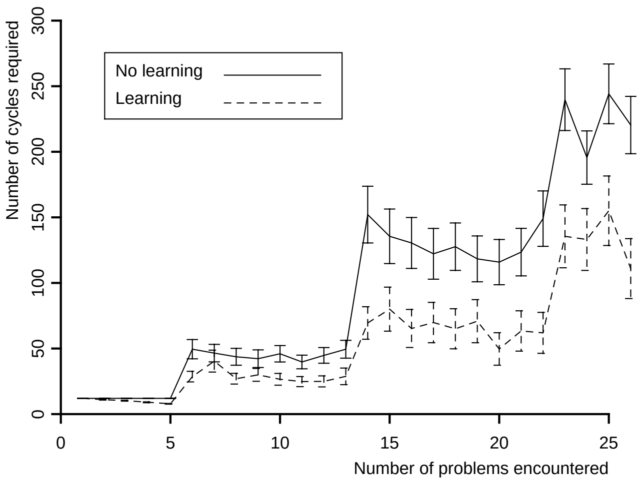

Figure 1. Number of cycles required by Icarus to solve a blocks-world task as a function of the number of training problems, averaged over 200 runs, with order randomized within each difficulty class.

We presented Icarus with problems in increasing order of difficulty because we believed this training regimen would lead to better learning. Our intuition was that, because the system would be more likely to solve simpler problems, it would be more likely to acquire skills and concepts that would prove useful in more complex ones encountered later. However, this hypothesis seemed worth testing experimentally, so we carried out another study with this in mind. In this case, we held back the four eight-step tasks for testing, and let Icarus learn only from the 22 simpler problems.

We examined three conditions, one in which (as before) problems were ordered randomly within their difficulty class, one in which they were ordered randomly without this consideration, and one in which no learning occurred. Again we averaged the required number of cycles over 200 different runs and, in this case, over the four test problems. As expected, the condition with no learning fared far worst, taking 230.16 ± 11.28 cycles. However, the skills acquired from problems ordered by difficulty took 140.13 ± 12.31, whereas those learned from randomly ordered training tasks took only 118.91 ± 11.62. Presenting simpler problems earlier cer-

[3] We ignored tasks with an odd number of steps, since these start or end with a block in the air. Also, we considered only problems in which the objective was a fully specified state.

tainly did not help ICARUS learn more rapidly, and may even have slowed the system down.

To understand better the factors at work, we repeated the random order condition with fewer training problems, again testing on the four eight-step tasks. When trained only on the five two-step problems, the average over 200 runs was 201.11 ± 11.09 cycles, and when the system learned from both these and the eight four-step problems, the average was 144.43 ± 11.05 cycles. Thus, ICARUS shows steady improvement with experience, suggesting that it acquires skills and concepts even from relatively complex training problems.

7. Related Research

The use of background knowledge to support learning has a long history within both AI and cognitive science. Research on explanation-based learning often aimed to improve efficiency on problem-solving tasks and combined experience with a domain theory to create new cognitive structures. Some work focused on the acquisition of search-control rules to guide problem solving, but other efforts instead dealt with the construction of macro-operators from primitive operators (e.g., Iba, 1988; Mooney, 1989). Our approach to learning hierarchical skills comes closest to the second tradition, since both involve composing knowledge elements into larger structures. However, ICARUS adapts this idea for the creation of skill hierarchies, whereas earlier methods produced flat macro-operators that contained less structure than the original knowledge base.[4]

ICARUS also has similarities to other cognitive architectures that incorporate varieties of explanation-based learning. For example, Laird, Rosenbloom, and Newell's (1986) SOAR revolves around a problem solver that proceeds until the system encounters an impasse, in which case it creates a subgoal to resolve it. This resolution may require search and take some time to produce the information necessary. Once the impasse has been handled, SOAR creates a *chunk* that encodes a generalized explanation of the result in terms of the original goal context. Intermediate steps from the solution are lost, but the chunk lets the system sidestep similar impasses in the future.

Anderson's (1993) ACT-R employs a related mechanism, called *compilation*, that creates new production rules from ones that are involved in the same reasoning chain. This scheme produces very specific rules that replace variables with the declarative elements against which they matched, rather than forming generalized structures, as do ICARUS and most systems that learn macro-operators. In fact, our approach is much closer to the composition process that played a role in earlier versions of ACT. The PRODIGY architecture (Minton et al., 1989) can invoke a similar mechanism to learn macro-operators from traces of means-ends problem solving. However, neither framework produces hierarchical structures or demonstrates the cumulative learning found in ICARUS.

A few researchers have built systems that support cumulative learning outside the context of problem-solving tasks. An early example was Sammut and Banerji's (1986) Marvin, which learns logical concepts that are composed of other, more basic, concepts. A human trainer presents the system with examples of increasingly complex concepts, ensuring that it has mastered each one before moving to the next. Pfleger (2002) describes another learning system that acquires hierarchical patterns in an unsupervised on-line setting. Like Marvin, it learns conceptual structures from the bottom up, so that more complex patterns are apparent after simpler ones have been acquired. Most recently, Stracuzzi and Utgoff (2002) report another learning system that operates in a cumulative manner.

A closer relative to ICARUS is Reddy and Tadepalli's (1997) system, which learns increasingly sophisticated goal-decomposition rules from a sequence of training exercises. Their system does not include an execution engine, but it does generate hierarchical plans and acquires its structures in a cumulative manner. Ruby and Kibler's (1991) SteppingStone also learns to solve more difficult problems based on solutions generalized from simpler ones, which it generates from a distribution and obtains through a mixture of problem reduction and exhaustive search. Benson's (1995) TRAIL incorporates a reactive control module that invokes learning when it reaches an execution impasse. The system learns from observing a teacher's actions and from experimentation, rather than from problem solving, and it acquires models for primitive actions rather than hierarchical structures, but its behavior is cumulative in that later learning depends on earlier experience.

8. Concluding Remarks

In the preceding pages, we presented ICARUS, a cognitive architecture for physical agents that uses stored concepts and skills, both organized in hierarchies, to recognize familiar situations and to control its behavior. We described a new module that supports means-ends problem solving on novel problems, along with a learning mechanism that produces new skills and concepts from traces of problem solutions. This method learns incrementally and in a cumulative manner, creating hierarchical structures that refer to others learned

[4] However, we have adopted Mooney's key idea that one should not chain off the preconditions of learned skills.

earlier. In addition, we reported experiments with the blocks world that showed such learning enables transfer to different problems and to more effective behavior.

Despite these advances, our work on cumulative learning in Icarus is still in its early stages. For instance, we should show its ability to learn hierarchical structures on domains other than the blocks world. These should include traditional problem-solving tasks such as the Tower of Hanoi, which has been the focus of many psychological studies. More important, we should study Icarus' behavior in dynamic domains that require integration of problem solving with reactive control. A prime candidate is the driving environment used to evaluate the architecture's categorization and execution modules (Choi et al., in press).

In addition, Icarus' methods for problem solving and hierarchical learning would also benefit from new capabilities. For example, the current system selects subgoals randomly when chaining off a concept definition, which means that it often needs to backtrack even when it has skills for each component problem. Extending the problem solver to avoid 'goal clobbering' would let it take better advantage of learned subskills. Nor can Icarus acquire recursive skills for tasks that involve regular structure, such as building towers in the blocks world. Analyzing relations among learned skills may provide this ability, which should let the system transfer learned knowledge to problems with more objects, as seems crucial for scaling to complex tasks.

We should also address directly the *utility problem*, which can actually produce slower behavior in systems that learn problem-solving skills. Our plans here involve storing, with each skill, its expected duration and probability of success, then using these numbers in execution and problem solving. Initial estimates would come from a skill's components but would be revised with experience in utilizing the skill. Combined with other extensions, this should give Icarus a more robust and effective approach to cumulative learning.

Acknowledgements

This research was funded in part by Grant IIS-0335353 from the National Science Foundation, by Grant NCC 2-1220 from NASA Ames Research Center, and by Grant HR0011-04-1-0008 from Rome Labs.

References

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

Benson, S. (1995). Induction learning of reactive action models. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 47–54). San Francisco: Morgan Kaufmann.

Choi, D., Kaufman, M., Langley, P., Nejati, N., & Shapiro, D. (in press). An architecture for persistent reactive behavior. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems*. New York: ACM Press.

Feigenbaum, E. A. (1963). The simulation of verbal learning behavior. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill.

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D., Etzioni, O., & Gil, Y. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, *40*, 63–118.

Mooney, R. J. (1989). The effect of rule use on the utility of explanation-based learning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 725–730). Detroit: Morgan Kaufmann.

Newell, A., Shaw, J. C., & Simon, H. A. (1960). Report on a general problem-solving program for a computer. *Information Processing: Proceedings of the International Conference on Information Processing* (pp. 256–264). UNESCO House, Paris.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Pfleger, K. (2002). *On-line learning of predictive compositional hierarchies*. Dissertation, Computer Science Department, Stanford University, Stanford.

Reddy, C., & Tadepalli, P. (1997). Learning goal-decomposition rules using exercises. *Proceedings of the Fourteenth International Conference on Machine Learning* (pp. 278–286). San Francisco: Morgan Kaufmann.

Ruby, D., & Kibler, D. (1991). SteppingStone: An empirical and analytical evaluation. *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 527–532). Menlo Park, CA: AAAI Press.

Sammut, C., & Banerji, R. B. (1986). Learning concepts by asking questions. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). Los Altos, CA: Morgan Kaufmann.

Utgoff, P., & Stracuzzi, D. (2002). Many-layered learning. *Proceedings of the Second International Conference on Development and Learning* (pp. 141–146).