CHAPTER
Two

Generalizing by Removing Detail:

How Any Program Can Be Created by Working with Examples

KEN KAHN

Animated Programs

S

R

L

Abstract

A long-standing goal of the programming by demonstration research community is to enable people to construct programs by showing how the desired programs should work on sample inputs. A major challenge is how to make the programs general. Heuristics and inference can generalize recorded actions on sample data in narrow domains but have yet to be much help in general purpose programming. This chapter describes a programming system called ToonTalk (Kahn 1996; 2000) that takes a different approach. In ToonTalk the programmer generalizes recorded actions by explicitly removing details. Children as young as six have constructed a wide variety of programs in this manner (Playground 2000).

2.1 Introduction

There is a very important interplay between the way in which programs are created and generalized in ToonTalk and the underlying model of computation. A program is executed as a collection of autonomous processes that communicate asynchronously in which the behavior of a process is specified by a set of guarded clauses. A clause is constructed by performing operations on a single sample data structure. To make the clause capable of operating on other data structures, the programmer needs only to remove details from the guard or conditional part of the clause.

ToonTalk is built on the idea of *animated programming*. Animated programs are not constructed by typing text or by constructing diagrams or stringing icons together. Instead, the programmer is placed as a character in an animated virtual world where programming abstractions are replaced by tangible analogs (see Figure 2.1). A *data structure,* for example, is a box whose holes can be filled with number or text pads, other boxes, birds, nests, and robots. *Birds* and *nests* are concrete analogs of send and receive capabilities on communication channels. A *robot* is a guarded clause that has been trained by the programmer to take actions when given a box. The thought bubble of a robot displays the guard or conditions that need to be satisfied before the robot will run. To generalize a robot, a programmer needs only to use an animated vacuum to remove details from the box inside the robot's thought bubble.

S

R

L

FIGURE **2.1**

Computational Abstraction	ToonTalk Concretization
Computation	City
Actor Process Concurrent object	House
Method Clause	Robot
Guard Method preconditions	Contents of thought bubble
Method actions Body	Actions taught to a robot
Message Array Vector	Box 4 2
Comparison test	Set of scales
Process spawning	Loaded truck
Process termination	Bomb
Constants	Numbers, text, pictures, etc. 1 9 9 7 ToonTalk
Channel transmit capability Message sending	Bird
Channel receive capability Message receiving	Nest
Persistent storage File	Notebook Pictures 1 Sally 2

Computer science terms and ToonTalk equivalents.

S
R
L

2.2 A Brief Introduction to ToonTalk

After thirty years of mixed results, many educators today question the value of teaching programming to children. It is difficult, and children can do so many other things with computers. Proponents of teaching programming argue that programming can provide a very fertile ground for discovering and mastering powerful ideas and thinking skills (Papert 1980). Furthermore, programming can be a very empowering and creative experience. Children who can program can turn computers into electronic games, simulators, art or music generators, databases, animations, robot controllers, and the multitude of other things that professional programmers have turned computers into.

Why do we rarely see these wonderful results from teaching children to program computers? The answer seems to be that programming is hard—hard to learn and hard to do. ToonTalk started with the idea that perhaps animation and computer game technology might make programming easier to learn and do (and more fun). Instead of typing textual programs into a computer, or even using a mouse to construct pictorial programs, ToonTalk allows real, advanced programming to be done from inside a virtual animated interactive world.

The ToonTalk world resembles a twentieth-century city. There are helicopters, trucks, houses, streets, bike pumps, toolboxes, hand-held vacuums, boxes, and robots. Wildlife is limited to birds and their nests. This is just one of many consistent themes that could underlie a programming system like ToonTalk. A space theme with shuttlecraft, teleporters, and so on, would work as well; as would a medieval magical theme or an Alice in Wonderland theme.

The user of ToonTalk is a character in an animated world. She starts off flying a helicopter over the city. After landing she controls an on-screen persona. The persona is followed by a doglike toolbox full of useful things.

An entire ToonTalk computation is a city. Most of the action in ToonTalk takes place in houses. Homing pigeonlike birds provide communication between houses. Birds are given things, fly to their nest, leave them there, and fly back. Typically, houses contain robots that have been trained to accomplish some small task. A robot is trained by entering his "thought bubble" and showing him what to do. This chapter focuses on how robots remember actions in a manner that can easily be generalized so they can be applied in a wide variety of contexts.

A robot behaves exactly as the programmer trained him. This training corresponds in computer science terms to defining the body of a method in

FIGURE **2.2**
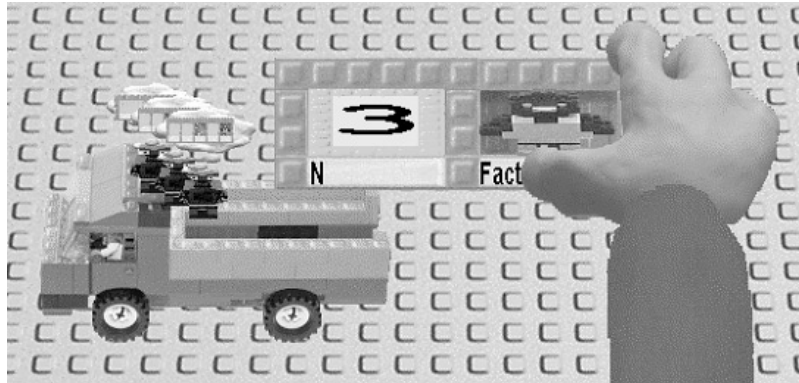
A truck being loaded with robots and a box.

an object-oriented programming language such as Java or Smalltalk. A robot can be trained to

- send a message by giving a box or pad to a bird;

- spawn a new process by dropping a box and a team of robots into a truck (which drives off to build a new house);

- perform simple primitive operations such as addition or multiplication by building a stack of numbers (which are combined by a small mouse with a big hammer);

- copy an item by using a magician's wand;

- change a data structure by taking items out of a box and dropping in new ones; or

- terminate a process by setting off a bomb.

The fundamental idea behind ToonTalk is to replace computational abstractions by concrete familiar objects. Even young children quickly learn the behavior of objects in ToonTalk. A truck, for example, can be loaded with a box and some robots (see Figure 2.2). The truck will then drive off, and the crew inside will build a house. The robots will be put in the new house and given the box to work on. This is how children understand

trucks. Computer scientists understand trucks as a way of expressing the creation of computational processes or tasks.

2.3   An Example of Programming by Example

The ideal way to convey how a programming by demonstration system works is to demonstrate it. While a live demo would be easier to follow, I'll present two detailed examples here on paper. (A dynamic replay of these examples is available from www. toontalk.com/English/wwhes.htm.) The first example is very simple. The benefits of a general-purpose programming with examples system are most apparent, however, with nontrivial examples. Hence, the second example is complex. While a complex example is hard to follow, it has the advantage that readers probably will need to think to figure out how to construct an equivalent program in their favorite programming language. Because the example requires some thinking, I hope that it shows how it really helps to program in a concrete fashion.

As our first example, let's imagine that Sue is a six-year-old girl who is fascinated by powers of 2. Sometimes in bed she repeatedly doubles numbers in her head before falling asleep. In ToonTalk she's discovered that she can use the magic wand to copy a number and then drop the copy on the original. Bammer, a small mouse with a big hammer, runs out and smashes the two numbers together. She starts with *1* and after ten doublings has *1,024* (see Figure 2.3).

She gets tired of repeated copying and dropping and decides to train a robot to do this for her. Since robots only work on things in boxes, she takes out a box and drops a *1* in it (Fig. 2.4[a]). She takes out a fresh robot and gives the box to the robot.

She finds herself in the robot's thought bubble, and as she moves the computer's mouse, the robot moves. She trains the robot to pick up the magic wand and copy the *1* and drop it on the *1* in the box (Fig. 2.4[b]). Bammer smashes them so that a *2* is now in the box. Sue indicates that she's finished training the robot and leaves his thoughts.

To try out her robot, Sue gives him the box with the *1* in it again (Fig. 2.4[c]). This time the robot knows what to do. Sue watches as he copies the *1* and drops the copy on it. But the robot stops. He won't work on a box with a *2* in it. Sue knows that this is because his thought bubble contains a box with a *1* in it, so the robot will only accept a box with a *1* in it. He's just too fussy.
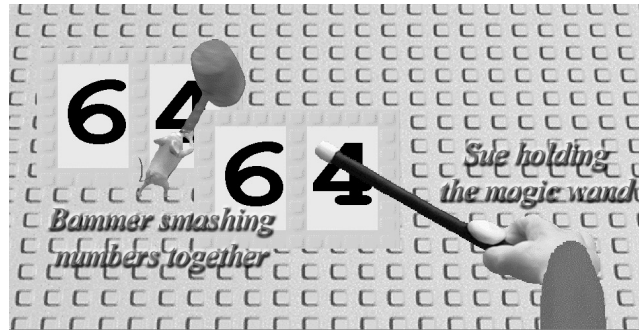
So Sue picks up Dusty the Vacuum and uses him to erase the *1* in the robot's thought bubble (Fig. 2.5[a]). Her robot is now thinking about a box

FIGURE **2.3**

*Manually doubling numbers by copying.*

with any number in it. She has generalized her program by removing detail—in this case by erasing the *1*.

Now she gives the box with a *2* in it to the robot, and he copies the *2* and drops it, resulting in a *4*. (Fig. 2.5[b]). He then repeats since the result still matches his thought bubble (Fig. 2.5[c]). Sue sits back, smiles, and watches as the number doubles over and over again.

Sally, Sue's twelve-year-old sister, is building a card game in ToonTalk. She has come to a point where she wants to sort the cards in a player's hand by rank. Without thinking very deeply about how to go about this, she creates a box with five numbers—*[3 9 6 4 2]*—and gives it to a robot. She then proceeds to train the robot to rearrange the numbers until she has *[2 3 4 6 9]*. When she gives the box to the robot, he repeats the recorded actions and sorts the box. She then creates another example box *[4 8 2 3 1]* and gives it to the robot, who rejects it because his thought bubble contains a copy of the original box *[3 9 6 4 2]* and this new box doesn't match. She then calls for Dusty the Vacuum and erases the numbers in the box in the robot's thought bubble. When she gives him the box *[4 8 2 3 1]*, once again she watches as the robot rearranges the box into *[1 4 3 2 8]*. A bit confused, she then gives the robot the box *[1 2 3 4 5]* and watches the robot turn it into *[2 5 4 3 1]*. While she thought she was training a robot to sort the numbers, she now realizes she had only trained a robot to permute five numbers so that the first number becomes the second number, the second one the fifth number, and so on.

ToonTalk does not contain program generalization heuristics. It does not try to solve the "data description problem" of figuring out what users mean when they manipulate objects on the screen. Instead, it has a small number
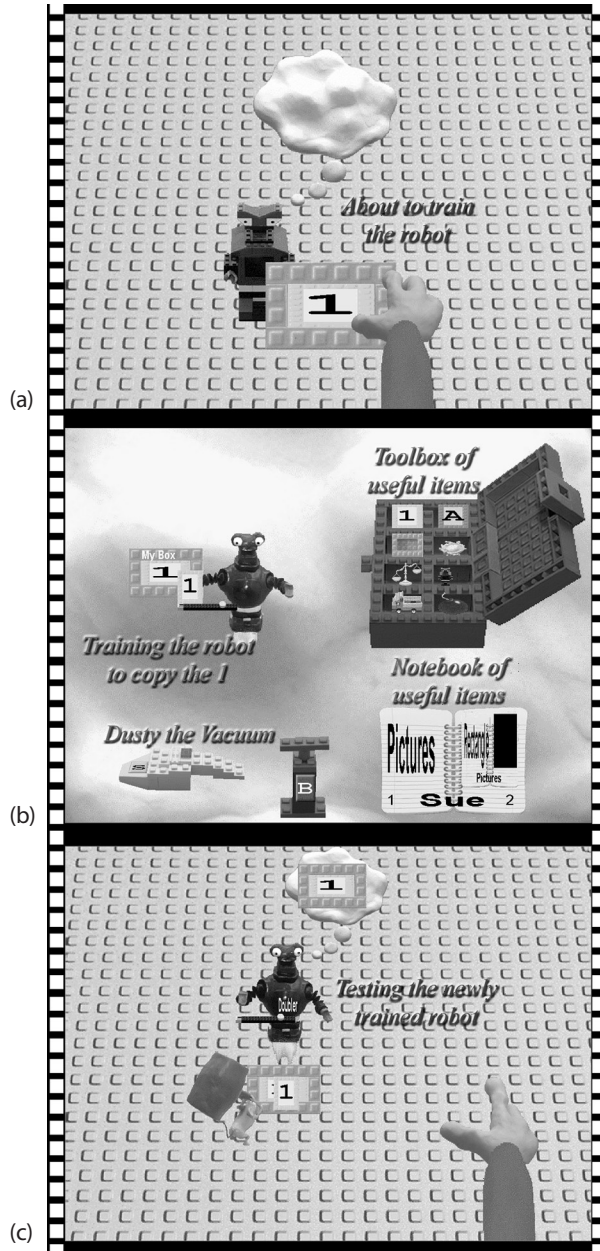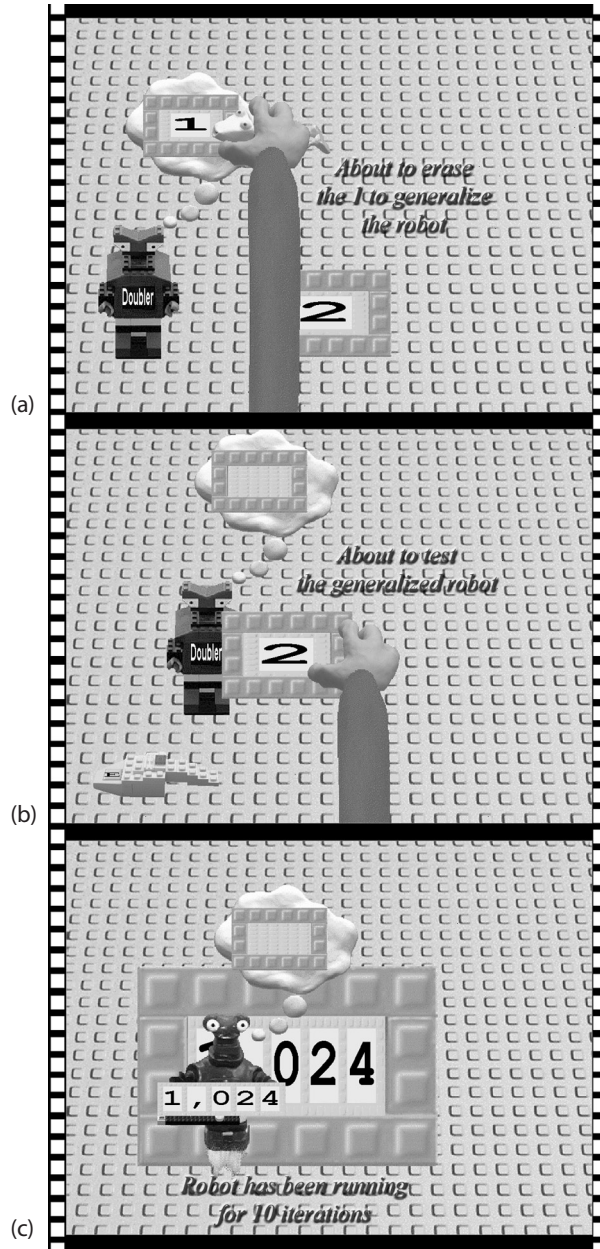
FIGURE **2.4**

Training a robot to double numbers.

S

R

L

FIGURE **2.5**

Generalizing and testing the robot.

TABLE **2.1**

Programmer's actions and how the robot remembers them.

Programmer's actions	Robot's recording of the actions
Took the *3* out of the box	Pick up what is in the *first* hole of the box
Set it down	Set it down
Picked up the *2*	Pick up what is in the *fifth* hole
Dropped it in the hole where the *3* was	Drop it in the *first* hole
Picked up the *9*	Pick up what is in the *second* hole
Dropped it in the hole where the *2* was	Drop it in the *fifth* hole
Picked up the *3*	Pick up the *first thing set down* (in step 2)
Put it where the *9* was	Drop it in the *second* hole
And so until the numbers were sorted	And so on

of very simple rules of generalization, and it leaves the task of weakening the guard conditions to the programmer. As the programmer trains a robot, the robot moves around, picks things up, drops things, and uses tools to copy or remove things. Robots remember the actions they were trained to do based on the position and history of objects. Robots ignore the path and timing of actions, labels (they are like comments in textual languages) and other details.

When Sally trained the robot to permute the box of five elements, she took the actions on the left side of Table 2.1. The robot recorded those actions as described on the right side of Table 2.1.

When she completed training the robot, his guard was the condition that his input be exactly the box *[3 9 6 4 2]*. After she erased the numbers, the guard only checked that the box contained exactly five numbers; the numbers could have any value. Clearly, relaxing the conditions didn't get her any closer to having a sort program. She could have trained 13!/8! robots to create a team that could have sorted any hand of five cards. The resulting program would only work with hands of exactly five cards. Instead, some other approach is needed.

She then thinks that maybe this is a job for recursion. If she could split the cards into two piles and sort those two piles somehow, she could merge the two piles by repeatedly taking the card from the pile that is showing the lower ranking card and putting the card down on a new stack. (In this scenario we'll see how she implements sorting by using the Merge Sort algorithm, but we could, of course, have seen her implement Quick Sort, Bubble Sort, or other algorithms instead. Also, we'll see her work with arrays when she could equally well have used lists or other data structures.)

S
R
L

Sally figures that the first thing she'll need to do is split the box of numbers in half. She might have been misled by working with concrete examples and thought she had to train a robot to break the box with five holes into a piece with three holes and a remainder with two holes. But then on the recursive calls she would have to train another robot to break up the box with three holes into boxes with two and one holes and also train yet another robot to break a box with two holes. By removing details, she'll be able to make each of these robots work for any numbers but the team will only be able to sort boxes with five, three, or two holes. Because ToonTalk is incapable of generalizing one of these robots into one that breaks boxes in half, she needs to train the robot to compute the size of the box, divide the size by 2, and use the result to break the box in half.

Sally knows that in ToonTalk you can find out the number of holes a box has by dropping the box on a blank number. (This is an instance of a general facility that uses blank or erased data types to express data type coercion.) She also knows that if she drops a box on a number, the box is broken into two pieces where the number of holes of one piece is equal to the number underneath. She now has a plan for how to train the robot to break the box into two equal parts.
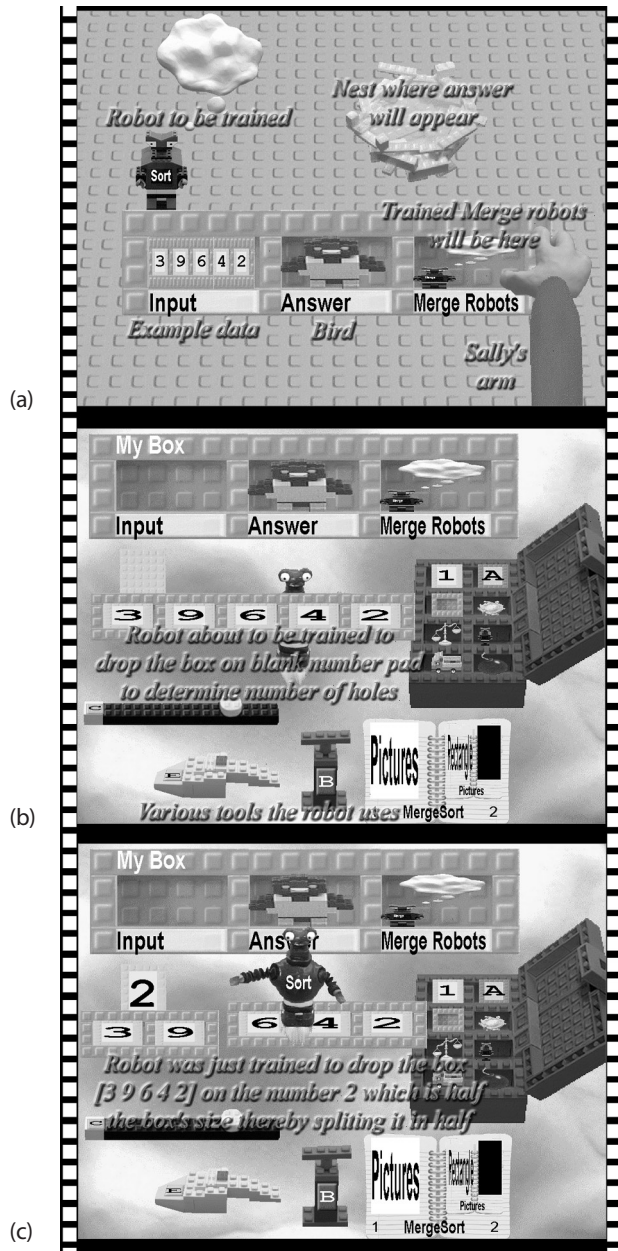
She'll need a robot that returns the sorted version of a box of numbers (see Fig. 2.6[a]). So she constructs a box whose first element is her first sample box *[3 9 6 4 2]* and whose second element is a bird who will be given the sorted version of the box when the computation terminates. Sally knows that the bird will then take the box to her nest. Since the *Sort* robots will need the help of *Merge* robots, she takes out a fresh robot and places it in the third hole. She plans to later replace that robot with the team of *Merge* robots after she has trained them. By placing a dummy robot in the hole, the *Sort* robots can now refer to the yet-to-be defined *Merge* robots. She takes out a fresh robot and gives him the box.

She enters the robot's thought bubble and trains the robot to create a blank number by taking out a number, erasing it, and dropping the list of numbers on it (Fig. 2.6[b]). The blank number changes to the size of the box. She then has the robot divide the resulting size by 2. (ToonTalk currently only supports integer arithmetic.) When she makes the robot drop the box on the number, the box splits into a part with two holes *[3 9]* and the remainder *[6 4 2]* (Fig. 2.6[c]).

Sally now removes the bird from the box and then makes a copy of the input box and puts the *[6 4 2]* box in the copy. She puts the *[3 9]* box in the original input box. She creates a box for the *Merge* process and puts the bird in it since the *Merge* process will produce the final result of the computation. She takes out two new nests and waits for eggs in the nests to hatch

S

R

L

FIGURE **2.6**

(a)

(b)

(c)

Training a Sort robot to split the problem in half.

S

R

L

into birds. She puts the birds in the holes labeled "Answer" in the *Sort* boxes. The nests she puts in the holes labeled "Sorted1" and "Sorted2" in the *Merge* box (Fig. 2.7[a]).

She trains the robot to take out a truck and use the magic wand to copy himself (and any future teammates), and to load the truck with his copy and the *Sort* box with *[6 4 2]* in it. She watches the truck drive off and takes out another truck. She puts a copy of the robot in the hole labeled "Merge" and the *Merge* box in the truck (Fig. 2.7[b]). She is now finished training the *Sort* robot and exits the thought bubble. The robot has now been trained to iteratively split the box in half, spawn another *Sort* process to work on the second half of the box, and spawn a *Merge* process to combine the results from the two *Sort* processes.

Sally is now ready to try out her sorting robot (Fig. 2.7[c]). She knows that she still needs to train more robots but wants to see this one work. She takes Dusty the Vacuum and uses him to erase the box in the hole labeled "Input" in the newly trained robot's thought bubble. By doing so the robot's "guard" is relaxed so that he'll accept any box whose first hole contains a box, second hole contains a bird, and third hole contains one or more robots.

Sally trained her robot to spawn a new *Sort* process to sort *[6 4 2]*. The robot then iteratively sorts *[3 9]*. She could have chosen to spawn two new *Sort* processes and let the current robot terminate instead. But there would be no advantage to doing so since iteration is tail recursion. In ToonTalk terms, the only difference is whether the team of robots continues to work in the same house (iteration) or a new house is built and the robots and box are moved there (tail recursion). She also trained her robot to spawn a *Merge* process to combine the results from the two *Sort* processes. Sally also needs to arrange for communication between these new processes so that the two *Sort* processes send their results to the *Merge* process. That is why she trained her robot to put a bird in the box used by each *Sort* process and the nests of the birds to be in the box used by the *Merge* process.

When Sally filled a truck with a box and a robot, the truck drove off. Since she was in the robot's thoughts, nothing was "returned" from these recursive calls. This ensures the generality of what the robot is being trained to do since it will not depend on the computation of any other program fragment.

One may be concerned that this scenario is complex—the robot is doing quite a large number of steps, and some are rather sophisticated. The alternative is even more difficult for most people: entering a symbolic encoding of *Merge Sort* with variables and parameters names. Here, the equivalent

FIGURE **2.7**
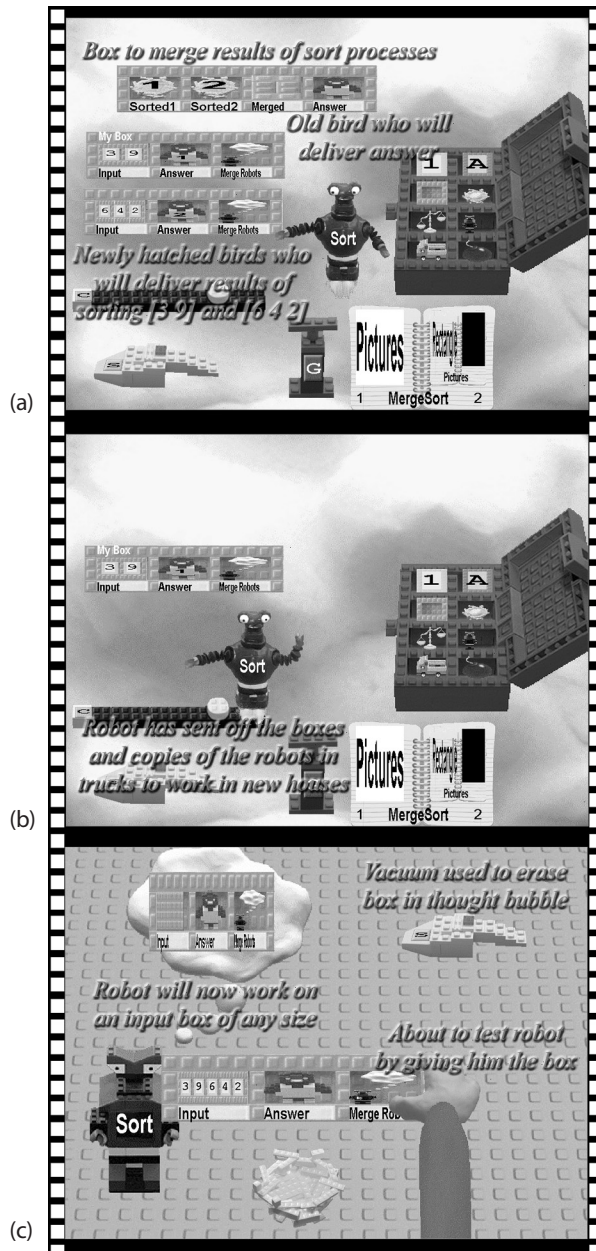
*Finishing the training of the Sort robot and generaliz-
ing it.*

structures and actions are concrete and tangible, thereby reducing the cognitive load for most people.

ToonTalk programmers generalize robots by erasing and removing things from their thought bubbles. The simplicity of this process relies on the fact that all conditionals in ToonTalk are depicted as a visual matching process. The story one might tell a student learning ToonTalk is that the robot's thought bubble displays the kind of box he's willing to work on. If he is given a box that differs from that, he'll pass the box along to another team member. But if he is given a box with more detail than he is "thinking about," then that is fine, and he'll accept the box. If a number pad, for example, is erased, then he'll be happy with any number in the corresponding location of the box he is given. This part of the conditional is simply a type test. If the number pad is removed, then he'll accept anything in that location. Comparisons of numbers and text are handled by pictorially matching with the tilt of a set of scales that indicates the relationship between the objects on its sides. If a robot expects an item in the box and sees a nest, he'll wait for a bird to put something on the nest. This is the essence of how processes synchronize.

Returning to our example, Sally gives the robot the box and watches him correctly reenact her actions. She watches as he splits off the *[6 4 2]* box and loads trucks to spawn the other processes. Her robot then iterates on the remaining *[3 9]* box. She watches the robot split the box and give the *[9]* part to a recursive process. But when she watches the robot try to sort the *[3]* box, she watches as he stupidly splits it into *[]* and *[3]* and then proceeds to work on the *[]* part. He then stupidly splits that into *[]* and *[]*. Sally realizes that she'll need more robots to help her robot with the easy problem of sorting one thing. She grabs her robot to stop him, sets him down, and takes out a fresh robot. She takes the box and replaces the *[]* box with a *[3]* box (see Fig. 2.8).

This new robot just takes the contents of the "Input" hole and gives it to the bird since the input is already sorted (Fig. 2.9). Sally then remembers that in ToonTalk it is good to be tidy and to destroy houses when a team of robots is finished. So she trains the robot to then pick up a bomb and set it off. This ends the robot's training. It also will stop the recursive process.

She now has a robot that will work with a box containing a *3* and another robot that will work on any box (but incorrectly if the box has only one hole). She calls for Dusty the Vacuum and erases the *3* in the box in the robot's thought bubble. By erasing the *3*, she has generalized the robot to work on any box containing a single number. If, instead, she had removed the *3* completely, then the robot would work with any box containing a single element regardless of its type.

FIGURE **2.8**
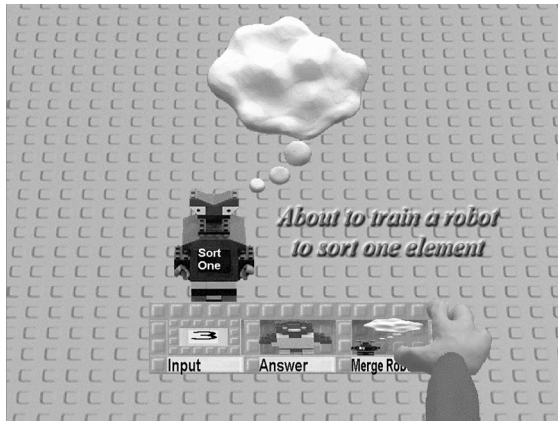
About to train a robot to sort boxes with one hole.

Sally drops the robot that will work on any size box on top of this new robot. The robot then moves behind the new robot, forming a team. If a box with one hole is being worked on, then the first robot will take care of it; otherwise, the second robot will do the work. She has now completed programming the *Sort* process robots and has only the *Merge* process robots left to do.

The training of the *Merge* robots illustrates an important programming technique of programming by demonstration—the use of derivative examples. Sally has run the *Sort* robots on an example, and they have created houses where dummy *Merge* robots have been given boxes to work on. These boxes are new examples that were generated by the *Sort* robots. Sally can go to one of these houses and start training a robot with the example box there. Then when a robot has been trained to do some of the processing on that box, she can run the robot until he stops. The robot has now generated another example box for training yet another robot.

She walks out of the house into a nearby new house where she sees a *Merge* box and a dummy robot. This is one of many houses that the *Sort* robots created by loading trucks. She wants to work on an example that covers lots of cases and modifies the box to become *[[4 6] [3 6 7] [1 2] bird]*. This represents the merging of the sorted records *[4 6]* and *[3 6 7]*, where the partial answer *[1 2]* has already been produced. The advantage of working with such an example is that in the process of training robots to accomplish this task, nearly all the different cases will be covered. It can be done with the

FIGURE **2.9**
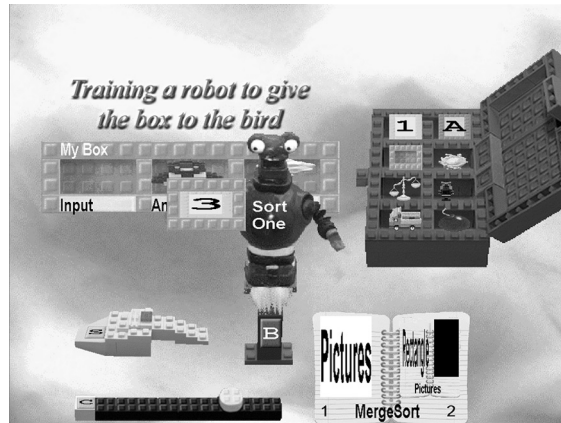
Training a robot for the base case of the recursion.

same number of robots as clauses in a traditional programming language and system.

She wants to train a robot to transform *[[4 6] [3 6 7] [1 2] bird]]* into *[[4 6] [6 7] [1 2 3] bird]*. But she knows that if she trained a robot to take the first element of the second box and move it to the end of the third box, the robot would do this regardless of the relationships between the numbers. Somehow she must make it clear why she is moving the *3* and not the *4*.

Sally knows she can use a set of ToonTalk scales to compare numbers. If the number on the left of a scale is larger than the one on the right, the scale tilts toward the left. Since she wants to compare the first numbers of each list, she realizes she'll have to put a scale between copies of those numbers. (Instead of copying, she could move the numbers and then move them back.)

She takes out a fresh robot and gives him the box she made (Fig. 2.10). She trains the robot to add three new holes to the box, place a scale in the middle hole, and use the Magic Wand to copy the *4* and put the copy to the left of the scale and put a copy of the *3* to the right. She watches as the scale tilts to the right. Her plan is to have another robot, whose guard (i.e., thought bubble) will check that the scale is tilted to the right and will move the *3* in that case. So she stops training this robot (Fig. 2.11).

Sally gives the box to her new robot and watches him reenact her actions. She then takes out a fresh robot and gives him the resulting box: *[[4 6] [3 6 7] [1 2] bird 4 > 3]*. She trains this robot to break the *[3 6 7]* into *[3]* and

FIGURE **2.10**
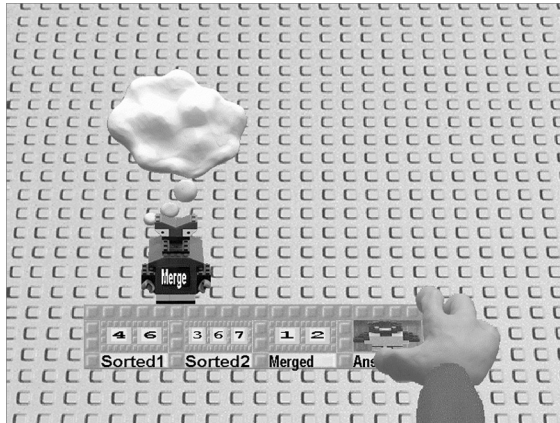
About to train the first Merge robot.

[6 7]. She puts the *[6 7]* box back and takes out the *[1 2]* box, joins the *[3]* box on the right, and puts the *[1 2 3]* box back (Fig. 2.12). This robot then removes the three temporary extra holes to restore the box for further processing.

Sally decides it is now time to generalize her two new robots. She decides the first robot should work with any three boxes followed by a bird and erases accordingly. The second robot should work with any three boxes, followed by a bird, followed by a number, a scale tilted to the right, and a number. She joins the robots into a team (Fig. 2.13).

She gives the *[[4 6] [6 7] [1 2 3] bird]* box to the *Merge* team. The box is transformed to *[[4 6] [6 7] [1 2 3] bird 4 < 6]*, and then the robots stop because the scale is tilted the other way so it doesn't match. She gives the box to a fresh robot and trains him very much like the previous robot. This robot differs only in that he moves the first number of the second, rather than first, box to the end of the third box. She generalizes him like the other robot and joins him to the team. The box is transformed to *[[6] [6 7] [1 2 3 4] bird]* and then to *[[6] [6 7] [1 2 3 4] bird 6 = 6]*, and then the team stops because the scale is balanced and none of the robots can match a balanced scale. Sally trains a robot to move both numbers to the end, generalizes him, and joins him to the team. The box is then transformed to *[[] [7] [1 2 3 4 6 6] bird]*. When the team iterates, the robot that adds the numbers and scale to the end of the box triggers an error message. Marty, the personification of

S
R
L

F I G U R E **2.11**
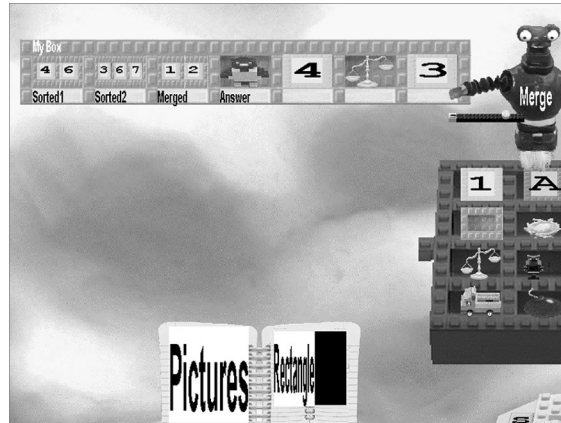
*Trained the robot to add the three new holes on the right.*

the error system, appears and gives her the warning that the *Merge* robot stopped because he was looking for a hole that a box didn't have. A robot needs to be trained to handle the case where the first hole contains a box without any holes. This robot needs to take the *[7]* and add it to the end of *[1 2 3 4 6 6],* give the result to the bird, and then set off a bomb because the task is completed.

After training this robot, Sally mistakenly believes she's finished and puts the teams in the correct holes and tries sorting some boxes. She tries *[3 2 1]* and she sees a bird fly to the nest with *[1 2 3]*—the right answer. But then when she tries *[3 9 6 4 2 1 2],* Marty appears and gives the same error message as before. To see what the problem is, she gets up, goes outside, and gets into her helicopter. She flies up and sees four extra houses. She lands near one and goes inside. She sees *[[2] [] [1] bird]* on the floor and realizes the problem is just that she didn't train a robot for the case where the second hole has a box with no holes. She trains a new robot much like the previous robot and adds him to the team.

Sally tries out her program again, and this time flies her helicopter over the city as it is running. She feels proud as she watches trucks driving to empty lots, houses being constructed, birds leaving some houses before they explode, while more trucks emerge from others. Sally has successfully constructed, tested, and debugged a parallel *Merge Sort* program.

S
R
L

FIGURE **2.12**
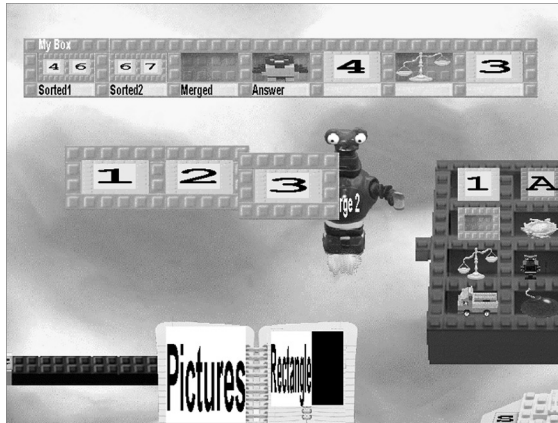
*Training the robot to add the first part of the box in
"Sorted2" to the end of the box in "Merged".*

2.4 Discussion

It might seem to be easier for Sally just to type in a parallel *Merge Sort* program in some textual language using variables rather than go through the process described here. One way to discover if ToonTalk is easier is to study real people using it and compare them with people using conventional programming tools. The Playground (2000) research project has begun such a study of children six to eight years old, and preliminary results are encouraging. Based on informal observations of about a hundred fourth grade students and feedback from many hundreds of beta testers and customers, we are confident that studies such as this will show a dramatic advantage to programming by example.

Program development is much easier in ToonTalk for several reasons. As many researchers in the field of programming by demonstration (Cypher 1993) have pointed out, people are generally better at working with examples than abstractions. Pygmalion (Smith 1993) and Tinker (Lieberman 1993) are two pioneering systems to support general-purpose programming by example. These systems were not able to eliminate abstractions completely, however. The need for a conditional test in Tinker, for example, was automatically discovered, but then the programmer needed to provide the

S
R
L

FIGURE **2.13**
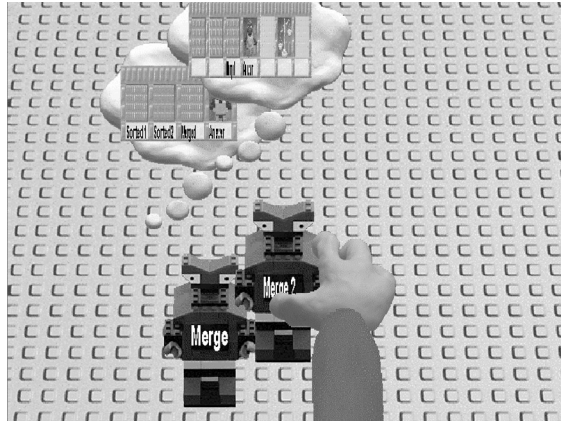
Joining the robots into a team after generalizing them.

predicate expression. When faced with multiple examples, both systems needed help from the programmer. Tinker asks the user which previous example is closest to the current one, for example. In Pygmalion, conditionals need to be introduced early in the process of demonstrating a program. In contrast, each ToonTalk clause (i.e., robot) is a self-contained unit whose conditional test is automatically generated by the system and relaxed by the programmer.

The task of supporting programming by demonstration is greatly simplified by the use of an appropriate computation model. ToonTalk has no nested conditionals, no complex conditionals (i.e., deep guards), no non-local variables, and no subroutine calls. It is nonetheless a very expressive high-level programming language. Each of these widespread programming abstractions interferes with the process of programming by example. Much of the power and simplicity of ToonTalk comes from the fact that once a robot starts working, it is just executing "straight-line" code without conditionals or procedure calls (though this straight-line code often includes process spawning and interprocess communication). In addition, any "variable references" are resolved to either the box that the robot was given or some local temporary variables that are concretized as something placed on the floor. Procedure calls interfere with a top-down programming style. If the subroutines aren't defined yet, then the system must, like Tinker, introduce

S
R
L

descriptions of data to represent the return values. These descriptions are not part of the programming language and add cognitive complexity. Procedures that modify data structures are even harder to handle. In ToonTalk, a procedure call is just a particular pattern of spawning processes and using communication channels to return results. A ToonTalk programmer frequently creates pairs of birds and nests and uses the birds to deliver computed values to nests. The nest, when used like this, is a "promise" or "future" value (Lieberman 1987). If a ToonTalk programmer needs to compute something before proceeding, then she splits the task between two robots, one to spawn the process that computes something and the other to use the result after it is received.

Pygmalion made the task of programming more concrete by letting the programmer manipulate iconic representations of programs and sample data. Tinker made things more concrete by letting the programmer manipulate sample data and descriptions of computations. ToonTalk makes the process of programming even more tangible by avoiding icons and descriptions and by providing concrete animated analogs of every computational abstraction. Everything in ToonTalk can be seen, picked up, and manipulated. Even operations like copying or deleting data structures are expressed by using animated tools like the Magic Wand and Dusty the Vacuum.

Stagecast Creator is the only other programming by demonstration system designed for children. Creator relies upon analogical representations. "Programming is kept in domain terms, such as engines and track, rather than in computer terms, such as arrays and vectors" (Smith, Cypher, and Tesler 2000, 78) In contrast, ToonTalk is programmed in computer terms, except those terms have "translated" to familiar and tangible objects such as boxes, birds, and robots. Consequently, ToonTalk is general and powerful, whereas "Creator emphasizes ease of use over generality and power" (Smith et al. 2000).

Special-purpose programming by demonstration systems derive much of their advantage from the fact that the task of extending or automating tasks in an application is very similar to the task of performing the task directly in the user interface. While ToonTalk is a general-purpose programming system, it also has this property. The same objects and tools used when training robots are also used by the programmer directly to accomplish tasks. Direct manipulation and recording are distinguished visually by whether the programmer is controlling a programmer persona (hand or person) or a robot. Furthermore, when controlling a robot the system displays the robot's thought bubble as the background instead of the usual floor. The same skills and way of thinking can be used in both modes.

2.4 Summary

According to Smith (1993), "The biggest weakness of Pygmalion, and all the programming by demonstration systems that have followed it to date, is that it is a toy system." ToonTalk is the first system in widespread use that enables programmers to construct general-purpose programs by demonstrating how the program should work on examples. Unlike its predecessors Pygmalion and Tinker, ToonTalk was not designed for computer scientists (Smith 1993) but for children and adults with no prior programming experience. Thousands of children have successfully used ToonTalk to construct a wide variety of programs, including computer games (Playground 2000), math and word manipulation programs, and animations.

Programming, however, is a cognitively challenging task. A programmer needs to design the program's architecture, break large tasks into manageable pieces, choose wisely among a range of data structures and algorithms, and track down and fix the all too common bugs once the program is constructed. Enabling the programmer to do these tasks while manipulating concrete data and sample input helps reduce some of the complexity. But the hard fun (Papert 1996) remains.

Acknowledgements

References

Cypher, Allen, ed. 1993. *Watch what I do: Programming by Demonstration.* Cambridge, Mass.: MIT Press.

Kahn, Ken. 1996. ToonTalk—An animated programming environment for children. *Journal of Visual Languages and Computing* (June).

———. 2000. ToonTalk Web site: www.toontalk.com.

Lieberman, Henry. 1987. Concurrent object oriented programming in Act 1." In *Object oriented concurrent programming,* ed. Aki Yonezawa and Mario Tokoro. Cambridge, Mass.: MIT Press.

S

R

L

———. 1993. A programming by demonstration system for beginning program-
mers." In

Papert, Seymour. 1980. Mindstorms: Children, computers, and powerful ideas. New
York: Basic Books.

———. 1996. The connected family: Bridging the digital generation gap. Longstreet.

Playground Research Project. 2000. Web site: www.ioe.ac.uk/playground.

Smith, David Canfield. 1993. Pygmalion: An executable electronic blackboard." In

Smith, David Canfield, Allen Cypher, and Larry Tesler. 2000. Novice programming
comes of Age. *CACM* 43, no. 3 (March). (See also chapter 1 of this book.)

S

R

L