Agent Problem Solving by Inductive and Deductive Program Synthesis

by

Harold Fox

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

Author ....
Department of Electrical Engineering and Computer Science
July 28, 2008

Certified by..
Howard E. Shrobe
Principal Research Scientist
Thesis Supervisor

Accepted by..
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Agent Problem Solving by Inductive and Deductive Program Synthesis

by

Harold Fox

Submitted to the Department of Electrical Engineering and Computer Science
on July 28, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

How do people learn abstract concepts unsupervised? Psychologists broadly recognize two types of concepts, declarative knowledge and procedural knowledge: know-what and know-how. While much work has focused on unsupervised learning of declarative concepts as clusters of features, there is much less clarity on the representation for procedural concepts and the methods for learning them. In this thesis, I claim that programs are a good representation for procedural knowledge, and that program synthesis is a promising mechanism for procedural learning. Prior attempts at AI program synthesis have taken a purely deductive approach to building provably corrent programs. This approach requires many axioms and non-trivial interaction with a human programmer. In contrast, this thesis introduces a new approach called SSGP (Sample Solve Generalize Prove), which combines inductive and deductive synthesis to autonomously synthesize programs with no extra knowledge outside of the program specification. The approach is to generate examples, solve the examples, generalize from the solutions, and then prove the generalization correct.

This thesis presents two systems, Spec2Action and HELPS. Given a logical specification, Spec2Action determines the relations to change to perform simple operations on data structures. The main part of its task is to uncover the recursive structure of the domain from the purely logical input spec. HELPS generates sequential programs with loops and branches using STRIPS actions as the primitive statements. It solves generalizations of classic AI tasks like BlocksWorld. The two systems use SAT solving and other grounded reasoning techniques to solve the examples and generalize the solutions. To prove the abstracted hypotheses, the systems use a novel theorem prover for doing recursive proofs without an explicit induction axiom.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist

Acknowledgments

I want to thank my advisor, Howard Shrobe. He gave me the freedom to pursue this research, despite my difficulty at articulating its vision and purpose at the beginning. He always let me work on the problems that I thought were important in the long run, even when they seemed odd and unrelated to my long term objectives.

I also want to thank my colleagues, Max Van Kleek, Gary Look, Alice Oh, Aaron Adler, and Jacob Eisenstein. They were always there to provide help and psychological support during the arduous task of finishing the system and writing the thesis.

I want to thank my thesis committee, Gerald Sussman and Patrick Winston, who were always able to provide a breath of fresh air and a dose of reality.

I finally want to thank my parents Siegrun Freyss and Charles Fox. They were very supportive of my decision to pursue a PhD in artificial intelligence, a leap into an unknown, but fascinating area from my undergraduate study in applied math and my brief career at an internet start-up.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

What is procedural knowledge? Imagine a chef who never graduated high school but who has been working in restaurants for thirty years. Then consider her apprentice, who graduated from the top cooking school, but who has never had any practical restaurant experience. Despite the schooling difference, the experienced chef can prepare meals much more efficiently and consistently than the apprentice. How could one represent the master's knowledge? Each kitchen situation is unique with an uncertain flow of orders each night. Somehow the master has learned general procedures and strategies for interacting with the world that are not explicitly written down in all of the apprentice's books and classes.

How could a computer represent procedural knowledge? How could it learn procedural concepts in a hands-on way by interacting with the world, reflecting on its experience, and generalizing solutions to day-to-day problems? This thesis will consider programs as a promising form of procedural knowledge. Programs are attractive, because they are provably correct, widely applicable, and robust to certain types of failure. Specifically, the thesis focuses on program synthesis as a core mechanism for procedural knowledge learning. The key thesis contribution is a new philosophy for program synthesis called SSGP for Sample Solve Generalize and Prove. To synthesize a program, an SSGP system generates examples and uses the examples to hypothesize general solutions to a program specification. Using SSGP, two systems have been developed, Spec2Action and HELPS, that can synthesize programs with considerably less knowledge and more autonomy than prior artificial intelligence synthesis systems which relied entirely on deductive synthesis.

1.1.1 Procedural Knowledge vs. Declarative Knowledge

Concept learning is a core area of cognitive science research. A concept, vaguely defined, is some symbolic entity of knowledge, often referred to by a word, such as "chair" or "fire". How do humans segment the vast, undifferentiated stream of sensory input and bodily impulses into discrete concepts? Humans certainly learn in a consistent way, because they are able to acquire language. Babies get very few examples of each word learned side-by-side with the word's referent, so it is generally accepted that humans do a considerable amount of unsupervised concept acquisition. How and why these concepts are learned and represented is a fundamental question in human psychology.

Concepts, and knowledge in general, can be divided into two types, declarative or descriptive knowledge and procedural knowledge: "know-what" and "know-how" [1]. Declarative knowledge represents concepts according to their observed features. A declarative representation of a chair would be an object consisting of a seat, legs, and a back in the chair configuration. Procedural concepts are represented by their role in accomplishing certain goals and tasks. A procedural representation of a chair is something to sit on.

Many cognitive architectures (systems seeking to model the complete human mind) explicitly model the two types of knowledge and their acquisition. Anderson, a cognitive scientist and computer scientist, implements the two structures explicitly in the Act-R cognitive architecture [2]. The system acquires procedural knowledge through "learning by doing". As it solves problems, it generalizes from its solutions and memorizes them for future use.

In computer science, there have been several interesting frameworks for the unsupervised learning of declarative concepts. Unsupervised clustering algorithms, Latent

Dirichlet allocations, and version spaces are some examples [5, 23]. On the other hand, there is generally much less clarity on the proper representation for procedural knowledge and the theoretical frameworks for learning it.

However, an argument can be made that procedural knowledge is more fundamental in human concept learning and knowledge representation. Social scientists who have studied education have discovered that humans learn concepts most effectively when they acquire those concepts in the process of accomplishing some goals that are relevent to them. This is "learning by doing" in contrast to "learning by studying." John Dewey, a progressive philosopher in the early 20th century, is an early and outspoken critic of disciplinary, fact-based education. He advocates natural learning and a non-adversarial approach to school [8].

Schank, a computer scientist turned education researcher, takes this idea further [29]. Students learn most effectively when they have a goal in mind. When they are motivated by the right goals, they naturally acquire the concepts they need to learn. In this theory of mind, the learning process consists of integrating one's current goals with past goals, and generalizing the common parts of the solutions. Schank calls these commonalities scriptlets, small strategies that participate in solving a general task. Schank's computer science work is in case-based reasoning, a problem solving technique similar to the caching and basic generalization of STRIPS plans.

1.1.2 Programs as Procedural Knowledge

There are several good potential representations for procedural knowledge. What makes programs unique and the focus of this thesis is that they are both provably correct and widely applicable. An alternative representation is a cached plan. The original STRIPS planning system stores the solutions to all of the problems it solves. Then, if one of these problems comes up as a sub-problem of a subsequent problem, the stored solution is substituted and the sub-problem is solved with no further reasoning. These cached plans are called macro-operators, since they are sequences of primitive operators. Cached plans are the representation for procedural knowledge in Act-R, Soar, and case-based reasoning [17, 2, 28].

Cached plans are valuable, because they are provably correct. Assuming that the world behaves as modeled by the STRIPS actions, if a cached plan is applied to a situation where it is applicable, then the subsequent state of the world is guaranteed to be exactly as desired. The problem with cached plans is that each one only applies to a narrow situation. A plan to clear a block A with two blocks on top of it could not be used if there are three blocks on top of A.

Another good procedural knowledge representation is a policy, acquired by reinforcement learning as an MDP or a POMDP [16]. Policies are decision procedures that can apply to any situation in a particular domain. The correct decisions for particular states are learned by maximizing utility on particular examples and generalizing. While policies are robust and widely applicable, they are not provably correct. That is, since a policy is learned on data and depends on various statistical assumptions, if a policy is moved into a new domain, there is no way of knowing if the policy will be successful or not. In other words, when a policy makes a decision, it cannot provide good reasons as to why it made that decision and what assumptions that decision was based on.

So, uniquely among these representations, a program constructed by program synthesis is both provably correct and widely applicable. Consider the problem of stacking block A directly onto block B shown in Figure 1-1. This problem is specified in Figure 1-2 with a program spec consisting of a start formula (assumptions about the world before the program is run) and a goal formula (the desired state of the world).

The specification is shown in Figure 1-2.

The Start specification states that the *on* relation is anti-symmetric, transitive, and that for all blocks x and y in one column, either x is on y or y is on x. Using STRIPS actions as the primitive operations, the program to solve this is in Figure 1-3

The program means: while there is something on A, move the top of the stack on A to the table. Then, while there is something on B, move the top of the stack on B to the table. Then, move A onto B. This program works for any configuration

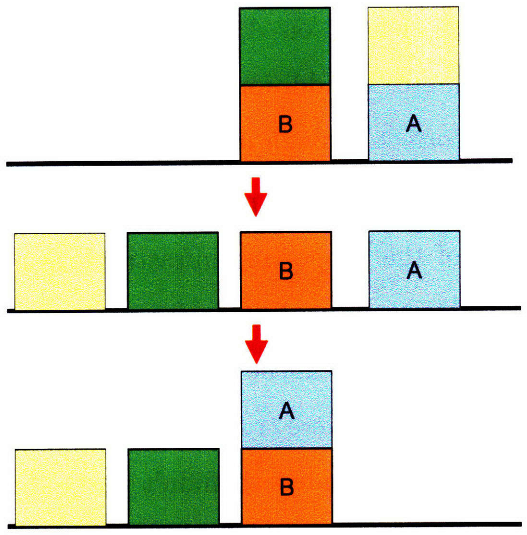

Figure 1-1: Operations to stack A directly onto B

$$\begin{aligned}
Start = \quad & \forall x \forall y\ (on\ x\ y) \Rightarrow \neg(on\ y\ x) \\
\wedge \quad & \forall x \forall y \forall z\ (on\ x\ y) \wedge (on\ y\ z) \Rightarrow (on\ x\ z) \\
\wedge \quad & \forall x \forall y \forall z\ (on\ x\ z) \wedge (on\ y\ z) \Rightarrow (on\ x\ y) \vee (on\ y\ x) \\
\wedge \quad & \forall x \forall y \forall z\ (on\ z\ x) \wedge (on\ z\ y) \Rightarrow (on\ x\ y) \vee (on\ y\ x) \\
\wedge \quad & \neg(on\ B\ A) \\
\wedge \quad & \neg(on\ A\ B) \\
Goal = \quad & (on\ A\ B) \\
\wedge \quad & \forall x\ (\neg(on\ A\ x) \vee \neg(on\ x\ B))
\end{aligned}$$

Figure 1-2: Specification to stack A directly onto B

```
1 : if $\neg(on\ A\ B)$
2 :   while $\exists x\ (on\ x\ A)$
3 :     $(moveTable\ x)$ : $x$ such that $(on\ x\ A) \wedge \forall y\ \neg(on\ y\ x)$
4 :   while $\exists x\ (on\ x\ B)$
5 :     $(moveTable\ x)$ : $x$ such that $(on\ x\ B) \wedge \forall y\ \neg(on\ y\ x)$
6 :   $(moveBlock\ A\ B)$
```

Figure 1-3: Solution to block stacking problem

of blocks. Using cached plans, there would have to be separate plans for when there is one element on B and one element on A, two elements on B and one on A, one element on B and two on A, etc. In addition to its applicability to many situations, a program is also provably correct through its construction by program synthesis.

When program synthesis operates on a program spec, it produces more than the code. It also produces all of the supporting infrastructure needed to know why a program is correct and what it accomplishes with each step. This infrastructure consists of invariant statements that can be counted on to hold at each step of the program if the starting assumptions were true. The invariants for the block stacking program are shown in Figure 1-4. (Readers shouldn't worry about the meaning of the logic or the theorem proving mechanisms at the moment. All of the representational and logical details are explained in later chapters.)

Given an invariant on one line combined with the action on the next line, the computer can automatically prove that the next invariant is true. Stepping through the whole program from the start to the end, the invariants comprise the proof that the program is provably correct.

When any program synthesis system produces a program, it also produces a correctness proof as a byproduct. Proof of correctness is the only way a program search knows it has a valid answer. The correctness proof has another benefit besides checking the produced program. It provides a continual consistency run-time check during the execution of a program. If the world ever changes in an unpredicted way, the program executor or controller will notice the change when the current statement invariant does not hold. In that case, the executor can trace backward through the program statements until it finds an invariant that does hold. The executor can then restart program execution at that point.

Because of the unique properties of provable correctness and wide applicability, this thesis investigates programs as a representation for procedural knowledge and program synthesis as the key mechanism in procedural knowledge learning. In the remainder of the thesis, it will be assumed that a general program specification has been provided a priori. However, a fully autonomous program learning agent would

```
$Start$
1 :  if $\neg(on\ A\ B)$
  $\neg(on\ A\ B)$
2 :      while $\exists x\ (on\ x\ A)$ {
      $S\ :=\ x$ such that $(on\ x\ A)$
      $T\ :=\ x$ such that $\neg(on\ x\ A)$
        $\exists x \in S\ \ \forall y\ \neg(on\ y\ x)$
3 :          $(moveTable\ x)$ : $x$ such that $(on\ x\ A) \wedge \forall y\ \neg(on\ y\ x)$
        $\exists x \in S\ \neg(on\ x\ A)$
        $\forall x \in T\ \neg(on\ x\ A)$
          }
$\forall x\ \neg(on\ x\ A)$
4 :      while $\exists x\ (on\ x\ B)$ {
      $S\ :=\ x$ such that $(on\ x\ B)$
      $T\ :=\ x$ such that $\neg(on\ x\ B)$
        $\exists x \in S\ \forall y\ \neg(on\ y\ x)$
5 :          $(moveTable\ x)$ : $x$ such that $(on\ x\ B) \wedge \forall y\ \neg(on\ y\ x)$
        $\exists x \in S\ \neg(on\ x\ B)$
        $\forall x \in T\ \neg(on\ x\ B)$
          }
  $\forall x\ \neg(on\ x\ B) \wedge \forall x\ \neg(on\ x\ A)$
6 :      $(moveBlock\ A\ B)$
  $(on\ A\ B) \wedge \forall x\ \neg(on\ x\ B)$
$(on\ A\ B) \wedge \forall x\ (\neg(on\ x\ B) \vee \neg(on\ A\ x))$
```

Figure 1-4: Invariants for the block stacking program

need to generate the spec as well as synthesize the program. Spec synthesis is a problem for future work. Further arguments for programs as procedural knowledge are contained in Schmid [31].

1.1.3 Program Synthesis by SSGP: Sample, Solve, Generalize, and Prove

The contribution of this thesis is a method for doing program synthesis from a spec without any interaction or any additional domain theory. The technique that enables this is called SSGP. It samples examples satisfying the start formula, solves them, generalizes the solution, and then proves the generalization correct. SSGP is used in two systems, Spec2Action and HELPS, that each synthesize different types of program. Prior systems that have sought to do program synthesis in artificial intelligence have used purely deductive synthesis.

Because the search space of potential programs is so large, generated examples can serve to guide the search, meaning that the synthesis systems can generate more sophisticated programs than prior art. They further require neither human interaction nor any guiding domain theory of lemmas and axioms. In particular, due to a novel logical representation and theorem prover, Spec2Action and HELPS can reason about synthesis problems without an explicit induction axiom. I believe that the ability to develop recursive structures and unbounded iterative procedures from first principles and examples is a more cognitively plausible model for how humans learn such structures and procedures in early learning.

1.2 The SSGP Systems

1.2.1 Introduction

To study program synthesis in AI and the efficacy of SSGP, two systems have been developed, Spec2Action and HELPS. Both systems use the same logical representations for worlds, and they share many common algorithms. For both systems, facts about worlds are represented with quantified formulas. These are very similar to first-order logic formulas with some subtle, but critical distinctions.

In general, it is assumed that a program operates on a world with a fixed set of objects and changing relations. Programs are specified with two quantified formulas, a start formula and a goal formula. A correct program applied to a world that satisfies the start formula will result in a world that satisfies the goal formula.

Spec2Action, given a program spec, determines the relations to change to satisfy the goal. Since the relations can be changed in any order, Spec2Action programs have no time component. The major concern with Spec2Action is determining the structure of worlds satisfying the start formula and the roles of the objects in them. For each program, Spec2Action determines what the roles are, hpothesizes a program on top of that, and then proves that such a program always works. The exercising examples for Spec2Action are simple operations on data structures, such as inserting into a linked list or inserting into a binary tree. A major subcomponent of Spec2Action is a system called Form2Grammar, which determines a recursive structure or grammar for worlds satisfying any particular start formula. Both Spec2Action and Form2Grammar use examples to guide their search for a program and grammar respectively.

The second synthesis system, HELPS, builds an iterative program using STRIPS actions. Since STRIPS actions have preconditions and they change many relations at a time, the correct sequencing of the actions is a critical concern. HELPS programs use loops, conditions, and let environments to control the actions. The exercise problems for HELPS are generalizations of classic AI problems such as BlocksWorld and other planning tasks.

Given a spec, Spec2Action determines "what to do," the relations that must change to take the start to the goal. HELPS, on the other hand, determines "how to do it," how to take the start formula to the goal formula using available actions.

1.2.2 Representations

To formalize program synthesis, we need precise definitions for environments, formulas, and programs. These are formalized fully in Chapter 2. When a program is

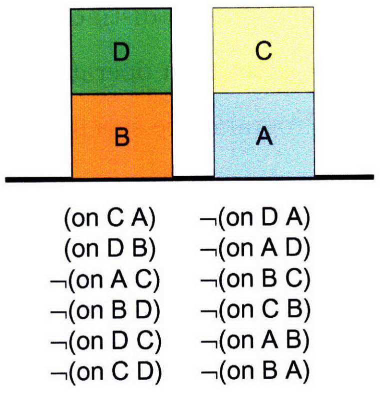

Figure 1-5: Sample environment for BlocksWorld

executed, it is applied to a concrete environment. An example environment is shown in Figure 1-5 for BlocksWorld. An environment consists of a set of objects O, as well as an instantiation of a set of relations $\mathcal{R}$ over O. In the case of Figure 1-5, $O = \{A, B, C, D\}$ and $\mathcal{R}$ is the binary relation $\{(on\ ?1\ ?2)\}$. To reason about sets of environments, we use quantified formulas. Quantified formulas are extensions of first-order logic with a few subtle, but important differences. An example formula is $\forall x \forall y\ \neg(on\ x\ y) \vee \neg(on\ y\ x)$. This means that for any two blocks x and y, if x is on y, then y is not on x. A formula precisely defines a set of environments for which it holds. Given any environment, it is always easy to check whether a particular formula is true.

Quantified formulas are used to specify programs: the starting condition and the goal condition. They specify the sets of environments that are valid start and goal states. Quantified formulas are also used to define actions: their preconditions and postconditions. The formulas specifying the block stacking problem are shown in Figure 1-2. The actions for moving a block to the table and for moving one block onto another are shown in Figure 1-6:

Programs are constructed using formulas and invocations of primitive actions. A program consists of five kinds of statements: primitive action calls, sequences of statements, if-then-else statements, while loops, and let environments. The sequence,

$$
\begin{array}{lrl}
(\mathit{moveTable}\ a) & & \mathbf{A}\ := x\ s.t.\ (\mathit{on}\ a\ x) \\
 & \textbf{precondition}: & \forall x\ \neg(\mathit{on}\ x\ a) \\
 & \textbf{postcondition}: & \forall x \in \mathbf{A}\ \neg(\mathit{on}\ a\ x) \\
 & & \\
(\mathit{moveBlock}\ a\ b) & & \mathbf{A}\ := x\ s.t.\ (\mathit{on}\ a\ x) \\
 & & \mathbf{B}\ := x\ s.t.\ (\mathit{on}\ b\ x) \\
 & \textbf{precondition}: & \forall x\ \neg(\mathit{on}\ x\ a) \\
 & & \forall x\ \neg(\mathit{on}\ x\ b) \\
 & & \neg(\mathit{on}\ a\ b) \\
 & & \neg(\mathit{on}\ b\ a) \\
 & \textbf{postcondition}: & \forall x \in \mathbf{A}\ \neg(\mathit{on}\ a\ x) \\
 & & \forall x \in \mathbf{B}\ (\mathit{on}\ a\ x) \\
 & & (\mathit{on}\ a\ b)
\end{array}
$$

Figure 1-6: Actions for BlocksWorld

if-then-else, and while statements work the same as they do in a language like C or Java. The conditions in the if statement and while statement are quantified formulas. The program in Figure 1-3 contains all of these structures.

1.2.3 HELPS

This thesis demonstrates three sub-systems, Form2Grammar, Spec2Action, and HELPS, each of which solves problems of differing complexity. Because program synthesis is an inherently undecidable task, none of these sub-systems have any guarantees of correct termination. However, if they do terminate, the solution is guaranteed to be correct. However, if a solution exists to a particular problem, there is no guarantee that these systems will find it. The systems are evaluated using a diverse set of exercise problems designed to reproduce the most difficult types of problems such a system might encounter in a real application.

The most advanced system is the STRIPS program synthesizer, HELPS, Hallucinated-Example Led Procedure Search. The synthesizer takes a start and goal formula as a specification and produces a complete program. The search works by applying various

$$\begin{array}{rl} \mathit{Start} = & \forall x \forall y\ \neg(<\ x\ y) \vee \neg(<\ y\ x) \\ \wedge & \forall x \forall y\ (<\ x\ y) \vee (<\ y\ x) \\ \wedge & \forall x \forall y \forall x\ \neg(<\ x\ y) \vee \neg(<\ y\ z) \vee (<\ x\ z) \\ \wedge & \forall x \in L\ \forall y \in L\ (<\ x\ y) \Leftrightarrow (\mathit{connected}\ x\ y) \\ \wedge & \forall x \in L\ \neg(\mathit{connected}\ x\ b) \wedge \neg(\mathit{connected}\ b\ x) \\ \mathit{Goal} = & \forall x \in L \cup \{b\}\ (<\ x\ y) \Leftrightarrow (\mathit{connected}\ x\ y) \end{array}$$

Figure 1-7: Specification for inserting an element into a sorted list

tactics to a program specification to generate sub-problems with smaller and simpler specifications. The search algorithm samples or hallucinates problem instances, solves the instances, and uses these instance solutions to decide which sub-problems look promising to pursue further. HELPS is described in Chapter 5.

1.2.4 Spec2Action

At present, HELPS can only solve problems with relatively short and simple goal formulas. The second system, Spec2Action, makes different assumptions about the final program format, and it solves a different set of problems. Reasoning about achieving a complex goal from STRIPS actions is difficult, because of the need to reason about side-effects and preconditions. With a complex goal formula, it is necessary to separate reasoning about relations that need to change from reasoning about how to use actions to change those relations. Spec2Action is a system that synthesizes programs to determine these relations to change. Currently, Spec2Action and HELPS are independent systems. However, the two systems are designed to complement each other, and future plans call for integrating them.

Spec2Action turns a program spec into an action schema, which describes the roles of the objects in the world, the relations to change on those roles, and the recursive construction of those roles on a world's grammar.

Consider the problem of inserting a block into a sorted list. This problem is shown in Figure 1-8. The specification is shown in Figure 1-7.

The start formula means that the elements are well ordered, and that the set L is

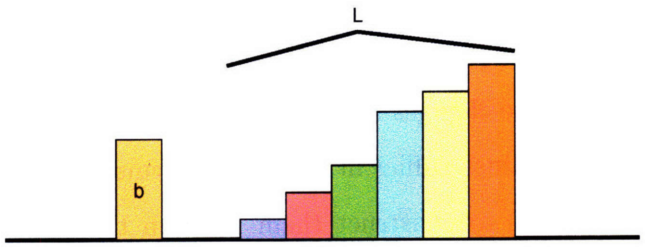

Figure 1-8: Inserting block b into a sorted list

$$\begin{array}{rcl} L & = & \{A, B\} \\ A & := & x \ : \ (< \ x \ b) \\ B & := & x \ : \ (< \ b \ x) \\ \mathit{Change} & := & (\mathit{connected}\ A\ b) \\ & & (\mathit{connected}\ b\ B) \end{array}$$

Figure 1-9: Relations to change (the action) for the insert sorted problem

sorted according to this order. The goal formula means that $L \cup x$ must be a sorted list. The goal formula is not specific about how the world should change. It only specifies that the final list should be sorted. Spec2Action takes this goal formula and produces the action solution in Figure 1-9.

The action specification divides the set L into two sets, A and B, pictured in Figure 1-10. A is the set of elements in L less than b and B is the set of elements greater than b. The elements of A are connected to b, and b is connected to the elements of B.

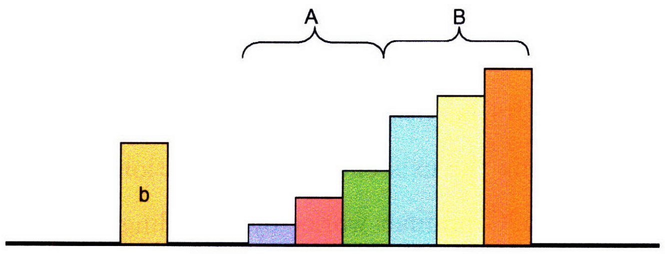

Figure 1-10: The list partitioned into sets A and B

In general, given a start formula and a goal formula, Spec2Action partitions the object set into disjoint sets $\{S_1, \ldots, S_n\}$with relations to change defined on these sets. Each relation to change changes in the same way for all elements of a particular set S_i. Spec2Action determines sufficient quantified formulas for the sets S_i, and it produces recursive formulas to construct them S_i from the grammar produced by Form2Grammar.

Simple operations on data structures have interesting actions, since they require uncovering the recursive structure of the data structure from the structure's logical description. Actions have been synthesized for a number of simple operations on data structures. Examples include inserting into a sorted list, inserting into a circularly linked list, and inserting into a binary tree.

Spec2Action uses SSGP. It synthesizes examples of the start formula and determines relations to change to satisfy the goal formula. Then, it generalizes from the solutions to these examples, trying to find the minimal action that works for all of the examples. Spec2Action then tries to prove that the chosen action works. It finds minimally sufficient conditions that the partition sets must satisfy, and it shows that such a partition can always be constructed out of an environment satisfying the start formula. Spec2Action is described in detail in Chapter 4.

1.2.5 Form2Grammar

Form2Grammar is the last subsystem, used by Spec2Action to prove the existence of a partition satisfying particular conditions. It takes an input formula and produces a recursive grammar that all environments satisfying the input can be decomposed into. For example, a grammar for the sorted list is shown below. The grammar is needed to produce a structure on environments satisfying a particular formula. The structure allows Spec2Action to find its partition by recursively processing the partitions of sub-environments and proving the existence of the partition by structural induction.

Like Spec2Action, Form2Grammar works by sampling, generalizing, and then proving the correctness of its generalization. It generates examples of the given formula, and then tries to build a minimal recursive structure to describe those examples.

It proves the generalization correct by finding minimally sufficient conditions for any particular non-terminal production. It then proves these minimally sufficient conditions by structural induction. Form2Grammar is described fully in Chapter 3.

All three SSGP subsystems work by a combination of inductive synthesis and deductive synthesis. They use examples to generate possible solution schemas. The systems then apply various automated reasoning algorithms to fill in the details of the schemas, so that the finished program works in all cases. This synthesis strategy is the main contribution of this thesis in comparison with previous work, which will be described and discussed in the following section.

1.3 Related Work

1.3.1 Foundations

The formalization of STRIPS planning originally came from the desire to produce a computer program that could respond flexibly and robustly to a wide range of situations. Newell and Simon's General Problem Solver(GPS), one of the earliest and most celebrated programs in AI, has the ambition to formalize a representation for all problems and all heuristic techniques for solving them [25]. Green equates problem solving with automatic theorem proving [15]. He shows how goals in the world can be achieved by treating actions as axioms, and repeatedly resolving them with the state.

This action representation led to STRIPS planning [11], the dominant formalisation used today in standard planning. Every primitive action has a precondition and a postcondition. The precondition defines the propositions that must hold for an action to be valid, and the postcondition defines the action's outcome. The planner uses search heuristics to efficiently find a sequence of primitive actions to make the goal true, given the initial starting state.

1.3.2 Generalized Planning

Abstracting Sequential Plans

There have been several attempts to create more expressive plans or to make planning solutions more reusable for later problems. One of the first extensions to the original STRIPS system is the concept of Macrops, macro operators [10]. Whenever a plan is made for a particular problem, all of the sub-sequences of the plan are converted into abstract, macro operators that can be used like ordinary, primitive actions. The preconditions and postconditions for this macro-action can be determined easily, and the macro-action can further be parameterized by removing any free variables. Macro operators are a simple, but powerful way to incorporate procedural knowledge into a problem-solving system.

Sussman extends generalized planning, and further articulates the vision of problem solving as automatic programming and debugging[38]. Sacerdoti introduces the idea of non-linear planning [27]. He analyzes the actions in a plan and determines which groups of actions can be run independently of each other in any order. This produces a plan that is a directed, acyclic graph. At any time, the plan executor may have a choice of a number of primitive actions to perform. Non-linear plans permit more modularity, and they allow the system to make a more informed decision about which sub-plans are useful to cache.

Contingency Plans

The need to cope with uncertainty when executing a plan in the real world has been the major driving force in the recent work on making plans more expressive and flexible. A contingency plan, introduced by Pryor and Collins, is a plan that can cope with uncertainty in the outcome of actions [26]. The world is assumed to be fully observable, but actions are non-deterministic. The set of possible world states that an action can change the world to is assumed to be known beforehand and is a small subset of all possible worlds.

Pryor and Collins extend partial-order planning [22] to plans with contingencies.

They define decision actions, which are capable of observing particular random variables at run-time. The values of these random variables determine the contingency, so they allow the plan executor to choose the proper path through the partial-order plan. In this algorithm, planning consists of heuristic search through the space of partial plans. At each step, a partial plan has an open condition satisfied, it has an unsafe link protected, or it has a random variable observed. A plan is complete when all conditions are satisfied in all contingencies, and there are no unsafe links.

Another variant of uncertain planning is conformant planning. In this case, the start state comes from a set of initial states, actions are non-deterministic, and the state is completely unobservable. The planner must produce a sequential plan that will work under all possible cases. This is essentially ordinary planning where the states are sets of possible worlds, elements of the power set of the original state universe. A set of states is called a belief state. Cimatti et al use binary decision diagrams to efficiently represent belief states and non-deterministic actions[6]. They then use symbolic model checking to explore all possible belief states that can reach the goal belief state. Symbolic model checking works for searching an exponential space of possibilities because it automatically exploits the wide range of symmetries inherent in most problems.

Bertoli et al extend symbolic model checking to partially observable planning [4]. Here, actions are non-deterministic and the initial configuration is one of a set of possibilities. However, at each stage, the state of the world is partially observable, so that the plan executor can choose alternate actions depending on the observations. The problem set-up is equivalent to contingency planning, but it explicitly separates the two sources of uncertainty in plan execution: uncertainty in the outcome of actions and uncertainty in the perception of the world. Bertoli et al represent a conditional plan of length n as an and-or tree of height n. An or node corresponds to all of the possible actions that can be taken at time t. An and node represents all of the possible states that the world could go to at time $t+1$ following the action at time t. To make a plan of size n, the problem is to find assignments of actions at each or node so that the and-or tree evaluates to true.

Bertoli et al represent belief states and plans as binary decision diagrams. As before, they use model checking to search all possible belief states, for which an and-or tree leads to the goal.

All of the above contingency and conformant planners are strong planners. That is, they work in all possible situations, and they do not assume any probability distribution over states and state transitions. Thus, the plans produced are provably correct, and so they fit into our requirements of provability and generality.

Plan Repair

Another way to deal with uncertainty in the world is to give the plan executor the capability of recovering from failure. When it executes a plan, it observes the world at each step. Whenever the observation does not match the plan's assumptions, the executor attempts to repair the plan. That is, it takes various steps to modify the existing plan so that it achieves the goal from the current state. Most planning algorithms can be modified to also perform plan repair [40]. Plan repair has been found to be a more efficient and stable way to recover from error than replanning from scratch [13].

Recursive Planning

Beyond contingency planning, a few researchers have attempted to construct plans with loops, called iterative planning or recursive planning. Plans with loops are considerably more difficult than plans with contingencies alone. First, there is the problem that the loop may never terminate. Proving that a loop does not terminate requires some form of automatic proof by induction. Second, it is difficult to reason about the preconditions and postconditions for a loop. Finding a precondition requires determining some type of loop invariant that will hold at the beginning and end of each loop iteration. Finding a postcondition requires finding a fixed point that remains true after any number of loop steps.

Deductive Synthesis over Plan Theory Manna and Waldinger is the first system in our survey that attempts true program synthesis [20]. Previously, these authors had developed a deductive tableau program synthesis system. This system synthesizes a program as the output of a theorem prover. The theorem prover is given an input condition, Start and a goal condition, Goal. It attempts to show by construction that there exists a function f, so that for every I such that $Start(I)$ holds, $Goal(f(I))$ holds. That is, the theorem prover constructs a functional program which satisfies the specification: $Start(.)$ and $Goal(.)$.

To use deductive program synthesis in planning, Manna and Waldinger create a plan theory, which allows them to do reasoning over functions of states. With each deduction rule, the plan becomes filled in, while the goal gets closer to True. Some rules introduce conditions to unify two plans with different assumptions. Others introduce action invocations applied to specific functions of the state. To perform recursive calls, there is an induction axiom and there are domain specific axioms to define a well-founded ordering on objects within states or on states themselves.

By treating general planning as functional deductive program synthesis, the authors must address many complex issues. One of these is the frame problem of situational calculus. How can a synthesis system reason about the facts of the world that remain unchanged after an action as well as the facts that do change? Another question is how can invariants be inferred that are strong enough to pass through an indeterminate number of recursive calls to a subroutine? In their system, the invariants must be axiomatized explicitly in the order relations.

Manna and Waldinger's system needs an extensive domain theory to produce even simple plans, such as clearing a block in BlocksWorld (shown in Figure 1-11). Thus, it is similar to an interactive theorem prover in which a human planner must provide most of the logical insight. In the conclusion, they argue: "We might speculate that human beings never completely prove the correctness of the plans they develop, relying instead on their ability to draw plausible inferences and to replan at any time if trouble arises...While imprecise inference may be necessary for planning applications, fully rigorous theorem proving seems better-suited to more conventional program

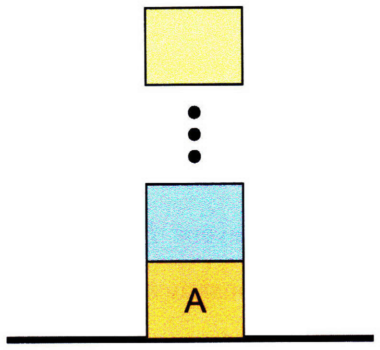

Figure 1-11: Clearing a block: the canonical generalized planning example

synthesis." Given the difficulties they have encountered, they conclude that fully general, provably correct program synthesis is too difficult to be practical.

This is overly pessimistic. First, functional programs and first-order logic are poor representations for program synthesis in AI. An iterative program captures the process of state change more naturally, it represents loops more explicitly, and it can keep track of necessary invariants at each statement more easily. Second, Manna and Waldinger try to generate a plan purely through deductive reasoning. This process quickly generates a very large search space of assertions to resolve. By combining inductive generalization with deductive reasoning, the proof search space can be focused in the most promising directions. Third, by assuming that a world consists of a finte set of objects with changing relations, the quantified formulas of Spec2Action and HELPS are simpler and easier to reason over. By using a general theorem prover, Manna and Waldinger lose many of the special-purpose algorithms and structures that can be devised for this problem.

Recursive Nonlinear Planning Ghassem-Sani and Steel demonstrate a working system, RNP, that is capable of automatically generating recursive plans[14]. They make the reasoning tractable by limiting the types of plan programs they represent. First, as HELPS does, they represent actions as STRIPS actions, thus eliminating the need for explicit frame axioms. Second, they assume a specific, yet common type of recursive subroutine. They assume there is a well-formed partial order on the objects in the world for a particular state of the world. They also assume a well-ordered reduction function *Reduce* and a base case *Base*, so that $\forall x\ \neg Base(x) \rightarrow Reduce(x) \ < \ x$. Plans consist of nodes in a partial order, which are primitive actions, conditions, procedures, and procedure calls.

Ghassem-Sani and Steel produce their plans using a partial-order planning algorithm: iteratively resolving conditions and protecting links. They include steps for adding cases and instantiating and performing recursive subroutines. That is, when they notice a goal $G(x)$ with a derived precondition of $G(Reduce(x))$, they instantiate a procedure node, whereby $G(Reduce(x))$ is solved with a recursive call.

Their logic is much simpler than Manna and Waldinger's plan theory, but it still enables the authors to automatically solve many important problems, such as clearing a block or reversing a list. However, too many details have been left out of the paper, so it is not clear what implicit assumptions they are making about the actions, order relations, and reductions. For example, it seems that the Base, Reduce, and $<$ operators must remain invariant across all recursive calls, yet the authors make no mention of this, nor do they indicate how they would automatically prove it. Thus, they may have surmounted the difficulties with Manna and Waldinger's plan theory by simply ignoring them.

Furthermore, in order for the planner to properly recognize a recursive precondition, it needs a well-informed domain theory including the ordering relation and reduction operator. Their block-clearing solution, for example, needs axioms about the $hat(x)$ operator (the block immediately above x) and the $clear(x)$ predicate (whether or not x is clear).

Temporal Planning Logic Stephan and Biundo also use deductive methods to generate a plan as a side effect of proving a theorem [37]. However, their system does not attempt to infer recursive plans or develop new loops from scratch. Instead, they assume that a schema for the problem solution has been provided from the user or from a plan library. Planning consists of refining the solution schema for a particular instantiation of the start and goal formulas. It basically consists of executing the abstract program schema in one particular world instantiation.

In this way, they separate the hard deductions from the easy deductions and assume that the hard deductions have been done off-line by the user with an interactive theorem prover. They argue that full plan generation including deducing recursive structures and induction proofs "is an interactive process with non-trivial inferences that in our opinion, which is shared by other authors as well, cannot be carried out in a fully automatic way."

Recursive Plans in Linear Logic Cresswell has the most recent and most extensive work on recursive planning in this survey [7]. He treats recursive planning as theorem proving in linear logic. Linear logic (Girard) is an interesting new representation for implication and inference. The logic introduces the notation $A \multimap B$. This means that the propositions in A can be used to show the propositions in B. However, each proposition of A must be used exactly once. The propositions of A are treated as resources that are consumed by B. For example, `have_coin` $\multimap$ `have_soda` means that the proposition `have_coin` leads to `have_soda`, but `have_coin` is no longer available. Duplicate propositions in the precondition are not redundant, so one could write `have_coin`, `have_coin` $\multimap$ `have_chips`, `have_soda`.

Certain common planning actions can be represented in linear logic quite naturally. For example, picking up a block `b` from on top of `a` can be written

`hand_empty`, `clear(b)`, `on(b, a)` $\multimap$ `holding(b)`, `clear(a)`

Conversely, putting down a block `b` onto block `c` can be represented as: `clear(c)`, `holding(b)` $\multimap$ `on(b, c)`, `hand_empty`. For a STRIPS action to be representable in linear logic, all of its preconditions must be negated following the action. Also, any postconditions that are not negations of a precondition must be positive propositions. In this way, two actions that consume different resources can be executed in parallel without interfering with each other. Linear logic deals with the frame problem and the hazards of conflicting actions very elegantly. It seems to be a much more suitable logic for reasoning about plans than the situational calculus used by Manna and

Waldinger.

Linear logic fits Green's idea of planning as theorem proving very directly and naturally. The theorem prover uses deduction steps of the form

$$\frac{S' \vdash G'}{S \vdash G}$$

This means that if G' can be produced by consuming S', then G can be produced by consuming S. Cresswell extends linear logic planning with some additional proof axioms that allow him to represent goals and conditions with disjunctions and universal quantifiers. He also creates induction axioms for a variety of recursive structures including towers of blocks, lists, and trees. As with deductive program synthesis, the plan is built directly out of each deduction step. The plan potentially contains conditions and recursive calls. Cresswell's theorem prover uses deduction steps in the form above with rewrite rules to equate equivalent functions of the state.

With this system, Cresswell is able to solve some interesting general problems such as reversing a tower, clearing a block, or moving blocks from one room to another with a robot. However, there are big differences between AbstrAct's approach and Cresswell's. First, actions of linear logic are a strict subset of STRIPS actions. Linear logic seems well-suited to tasks that have a clear flow of activity: one action directly consuming the result of the previous action. Such tasks are natural in manipulating physical objects in the world. However, it is less clear how linear logic could represent tasks such as sorting a list or setting the bits on a bit counter. Second, Cresswell takes a fully deductive approach to producing generalized plans. He has to produce induction axioms for each new domain that he studies. He also has to define a recursive structure for the object groups that he studies: towers, lists, trees, etc. And, he has to produce rewrite rules that allow him to equate recursive functions of the state.

Basically, Creswell, like the two other generalized planners, takes a knowledge-intensive, deduction-based approach to solving problems. He states, "Generally, the modelling of the problem in a suitable form in linear logic was found to be a difficult process...The difficulty involved in constructing reasonable formulations for the domains cannot easily be separated from the difficulty of solving them." In contrast, Spec2Action and HELPS combine exploration-based, inductive synthesis with deductive synthesis to solve problems with much less information provided a priori.

Distill The last three systems in this survey do not attempt full recursive or iterative planning. Instead, they focus on generating specific types of commonly occuring loops to make more generally applicable plans. Distill, from Winner et al, is a system to build a library of domain-specific planning algorithms [41]. That is, instead of finding one program a priori to provably solve all instances from a domain, they find a program that is capable of solving all previously solved instances. Their contribution is the Distill algorithm, which compactly combines these plan solutions into one large condition statement. This learned program can also acquire a limited type of loop. Whenever a solved instance has a sequence of identical, independent subplans, this subplan is generalized into a loop. For example, loading a set of packages from a truck to a warehouse is a set of tasks independent of each other that can naturally be put into a loop. Unstacking a tower of blocks is not independent, because the blocks must be unstacked in a particular order.

Distill has a similar strategy as AbstrAct for building a plan program: learning from a set of examples. However, Distill does not attempt to solve the whole planning problem from these few examples or do any sort of reaoning/theorem proving. So, its general applicability is restricted to what it can safely generalize from the individual examples it has seen thus far.

KPlanner KPlanner is another approach to generating plans with loops and conditions [18]. The demonstration example is chopping down a tree. The system performs the chop action until it observes that the tree is down. The key enabler for these plans is a planning parameter F, which often corresponds to the maximum number of loop iterations. The user provides a generating bound and a testing bound for the planning parameter. In the chopping problem, F is the number of chops needed to fell a tree. The generating bound is a small number, just large enough for the system to recognize an unrolled loop. The testing bound is a safety value to ensure that the completed plan will terminate in all cases.

With the generating bound, the planner finds a contingency plan that works under all circumstances. KPlanner then performs pattern matching, looking for the format of an unrolled loop, which it then rolls up. Finally, it tests the proposed plan against the testing bound. AbstrAct works like KPlanner in that it attampts to derive a general loop body from looking at example instances. However, AbstrAct works with more general problems with harder plans to prove. KPlanner problems are fundamentally the same as the ones given to the contingency planners above. The number of objects is known, but the actions and the initial state are non-deterministic. Thus, for example, KPlanner could not be used to produce a general plan for clearing a block.

Abstraction for Generalized Planning Srivastava et al present preliminary work using abstraction to do generalized planning [36]. Their approach is to combine a set of objects into a larger super-object, called a role, where all of the objects in the role satisfy the same unary predicates. For example, in BlocksWorld, the base object forms one role, the topmost object forms another role, and the middle objects form the third role. The authors then use a complicated set of rules to determine what meta-relations hold between roles in individual instances, and how those meta-relations change on the application of particular actions. By keeping track of the sizes of roles, the system is able to reason inductively over actions that move individual objects from one role to another. With these finite number of roles, the authors create a transition graph with primitive actions among all possible world abstractions. Planning then consists of finding a path in the graph from starting roles to goal roles and recursively finding conditions on the sizes of the roles at each step so that the path was correct. The process can infer loops similar to KPlanner by noticing when a particular path repeats itself. Then, the condition for a looped path is the union of the conditions for all numbers of cyles through that loop. Planning finishes when there are no more paths or when union of the conditions on the sizes of the roles at the start state becomes True.

Abstracting objects into sets and reasoning about the sets and the sizes of sets is a very promising direction. It forms a major component in many of Spec2Action and HELPS algorithms as well. In our systems, however, the roles are not fixed a priori, and they are discovered by looking at examples. Thus, the SSGP systems need less information, and are applicable to a wider range of problems. In particular, in the BlocksWorld representation, there is no top block and no base block. The only relation is $(on\ \ x\ \ y)$. Srivastava et al's system would not be able handle this situation.

1.3.3 Program Synthesis

Program synthesis or automated programming is a large and dynamic area of research in software engineering [12]. Many of the strategies from this field carry over to programs in the planning domain. Program synthesis is generally done as an interaction between the user and a reasoning system. The user provides the high-level direction for solving the problem, and the synthesis system fills in the details. The motivation is to have an efficient way to allow expert users to produce provably correct programs.

There are broadly three styles of program synthesis. Deductive synthesis generates a functional program as the byproduct of a resolution theorem proof. Schema-guided synthesis uses a program template and reasons over the synthesis problem to find constraints on the parameters of the template. It then tries to find concrete functions to meet those constraints. Inductive synthesis uses a set of example problems and tries to find a program that fits all of the examples. Pure inductive synthesis is different from the other two, because it does not seek a provably correct program. In fact, an inductive synthesis system generally does not even have any logical description of what "correct" means. It is essentially a learning task.

Deductive Synthesis

Deductive synthesis is a simple and elegant way to generate pure functional programs. Manna and Waldinger provide a readable and comprehensive overview of their system [21]. The system takes as input a start predicate, $S(.)$ and a goal predicate $G(.)$. The system then uses a theorem prover to prove the theorem $\exists P$ such that $\forall x \; S(x) \rightarrow G(P(x))$. The proving takes place over a 3-column deductive tableau. The tableau has assumptions in the first column, goals in the second column, and partial programs in the third column. Different rows are resolved together using standard resolution and unification techniques. There are simple rules for combining the program skeletons of each row. There is typically an induction axiom which creates a corresponding recursive program. The synthesis finishes when an assumption column becomes False, or a goal column becomes True. Manna and Waldinger have been able to synthesize some interesting programs such as factorial and a square root algorithm.

Schema-Guided Synthesis

While deductive synthesis is an elegant technique, it doesn't scale well to larger problems. In this case, to make synthesis tractable, users provide a skeleton of the solution, and domain-specific synthesizers can fill in the gaps. An important example is the Cypress system for generating recursive divide-and-conquer programs [32]. A divide-and-conquer program has three components: a base case for dealing with the simplest inputs, a splitting function for dividing an input into smaller pieces, and a combining function for merging the solutions of the previous steps. Cypress uses the problem specification to find minimally sufficient conditions that each of the components must satisfy. It then uses a set of domain-specific axioms to generate functions satisfying each of these conditions. An example problem it solves is MergeSort. The base case is an empty or singleton list. The splitting function is just splitting the list in two. The combining function is the merge operation. The merge operation is itself a divide-and-conquer operation, whereby the smallest element of two lists is split off and added to the merge of the rest of the two lists.

Smith et al combine Cypress and several other tactical synthesizers into the program synthesis package Kids [33]. Specware is the Kestral Institute's next generation follow-up system to Kids[35]. It consists of a language for the formal specification of programming problems as well as a way to define domain theories consisting of axioms and theorems about a particular class of problems. Part of Specware is a formal semantics for specifications that describes how specs can be combined and refined into operational programs. The user constructs a program by manipulating these specs. Each refinement of a logical spec into a more operational procedure produces proof obligations that are automatically proved by a theorem prover (Isabelle/HOL by default).

An important application of Specware is a theory for generating domain-specific planners called Planware. Based on examples from transportation scheduling, Planware contains theories and specifications for solving general classes of scheduling problems. When solving a specific scheduling problem, the Planware user invokes the tactics that they feel are relevant, and Planware produces a scheduling program specific to the user's constraints. On test examples from military logistics, the scheduling programs produced by Planware run significantly faster than a general-purpose scheduler. Planware illustrates the potential for program synthesis in problem solving. While there are difficult theorems to be automatically proven up-front, the resulting program works much better when applied to concrete examples. Any general thinking and reasoning that could be done has been factored and abstracted away into problems that only need to be solved once.

Meta-Amphion, from NASA Ames Research Center, is a similar knowledge-based software engineering system [19]. It combines a deductive synthesis system with a library of previously solved general problems and search tactics. The resulting system generates correct programs much faster than a deductive synthesizer alone.

Another NASA synthesizer is AutoBayes, a domain-specific engine that constructs an algorithm for learning the parameters of a Bayes Net graphical model. Algorithmically, parameter learning is an optimization problem, so AutoBayes analyzes the structure of the model specification to select the fastest, most appropriate optimization algorithm.

A final schema-based system is SKETCH, a synthesis plug-in designed to integrate more closely with existing programming languages such as C [34]. In this system, the user writes a program template, which includes the basic variables and expressions that need to be used, without specifying how they should be combined together. The user also writes a test function that is a simple, logically correct version of the function, potentially with an inefficient implementation. The test function serves as the specification, so the user does not need to think about writing abstract logical formulas. The synthesis problem boils down via syntactic sugar into a problem of finding a vector of constants to make the sketched program logically equivalent to the test program. The synthesizer iteratively guesses values of these vectors, and a model checker finds a counterexample input to make the hypothesized sketch function not equal to the test function. The system then finds new constants to fit every counterexamples seen thus far. The loop continues until the model checker agrees that the two functions are logically equivalent.

Deductive and schema-guided synthesis are both knowledge-intensive automatic programming methods. They rely on the user to provide the high-level intuition for solving the problem, and the synthesizer fills in the individual details that humans find tedious and error-prone. In contrast, this thesis investigates whether a computer is capable of learning this knowledge from scratch. So, while the SSGP systems in general work on simpler problems, they do so with much less guidance and certainty.

Inductive Synthesis

Inductive synthesis is a family of techniques to learn a program from a set of examples. In general, the goal is to find the smallest, simplest program that still fits all of the examples. Like version spaces, inductive synthesis is often touted as a means for computers to learn symbolic concepts. Inductive Logic Programming is one of the most popular techniques [24]. Given a set of example concrete relations (e.g. `(mother Jane Joe)` `(sister Mary Joe)`, `(aunt Jane Mary)`) and a target relation (e.g. `(aunt ?1 ?2)`), ILP tries to find a minimal set of Horn clauses to properly infer the target. There are many fast algorithms for doing ILP, and Horn clauses are a good knowledge representation: efficient to use, but expressive enough for many tasks.

A complementary technique is explanation-based learning [9]. Here, the system knows the whole dynamics of the universe encoded in a domain theory (e.g. all the moves of Chess). However, this domain theory may be too complex to reason over efficiently. Using a training example, the system finds the smallest set of rules in the domain theory to prove the example, i.e. the explanation (e.g. explaining a cerain arrangement of pieces that define a checkmate). From seeing many examples, the system learns simple rules that are useful in explaining the common things it may see. Acquiring these simple rules is a type of procedural concept learning. Indeed, explanation-based learning can be used to describe any problem where all the knowledge is available before-hand, but the goal is to turn that knowledge into simpler rules that are more useful.

Program synthesis by SSGP is thus a type of explanation-based learning. The world and its actions are fully specified in high-level logic, but they are too unconstrained a priori. SSGP uses examples to narrow the search for a provably correct program.

A popular application of inductive programming is programming by example. This is a technique application designers use to allow non-expert end-users to create simple programs. Users create examples of what they want their program to do, and the application uses an inductive synthesis algorithm specialized to the task at hand to find a match.

Another interesting approach to inductive programming is the L* algorithm of Angluin [3]. The algorithm learns a finite state automaton to describe a set of input strings. For training, the algorithm can query an oracle about any hypothesized input string. L* continually builds its state set and example set until all of the examples can be assigned to a state such that all of the examples assigned to the same state behave consistently thenceforth.

Schmid presents an inductive synthesis algorithm for the explicit purpose of learning a generalized plan from examples [30]. Each example is a STRIPS planning problem. She solves each one individually and combines the solutions into a generalized plan, potentially with recursion and conditions. This is the same approach that as SSGP, and her motivations for program synthesis as procedural knowledge learning are the same as the motivations of this thesis. In describing recursive schemas, she argues: "We believe that our approach mimicks certain aspects of human (novice) programmers." In another paper, she directly argues that general program synthesis out of specific examples should be the model for procedural knowledge acquisition in mimicing human intelligence [31].

HELPS goes beyond Schmid's recursive schemas, because in addition to generalizing a program, it reasons that it is correct. My view is that deduction of correctness from a high-level logical description of a problem is as important as bottom-up induction from examples, and the two approaches should complement each other.

1.4 Thesis Organization

Chapter 2 covers the representation for worlds and quantified formulas. It also covers some basic formula transformations. The next three chapters cover the three SSGP systems, Form2Grammar, Spec2Action, and HELPS. The simplest systems are covered first. This will allow a clear, measured introduction to each of the algorithms and strategies involved. Each system is evaluated independently by a small set of example problems, and so each of these three chapters has its own results section. Chapter 3 covers Form2Grammar, the system for discovering a recursive structure for worlds satisfying a particular logical formula. Chapter 4 covers Spec2Action, the system for converting a program spec into a more understandable action specification. Chapter 5 covers HELPS, the system for learning and proving general STRIPS plans with conditions and loops. Chapter 6 concludes with proposals for future work.

Chapter 2

Representations of Worlds, Formulas, and Problems

2.1 Worlds

This chapter makes precise all of the concepts that we will use in developing and explaining the SSGP systems, Form2Grammar, Spec2Action, and HELPS. Most of the definitions and theorems are straightforward and intuitive. First-time readers can skim the chapter and refer back to it to clarify any ambiguous concepts.

The representation of worlds and formulas is the same for all of the SSGP systems. A world, also called a state or an environment, contains all of the concrete information that a plan executor cares about at a particular point in time. Intuitively, a world is a finite network of objects and relations. Formally, a universe of worlds Ω is defined by a relation template set $\mathcal{R}$. Each relation template $R \in \mathcal{R}$ defines a type of relation between objects in the world. It is essentially a function that takes objects of the world as parameters and returns $True$ or $False$. The *arity* of a relation is the number of parameters that it takes.

For example, in BlocksWorld, there is one relation template (*on* ?1 ?2), which has arity 2. $\mathcal{R} = \{(\textit{on}\ ?1\ ?2)\}$.

Worlds can be described by quantified formulas, which are a straightforward extension of first-order logic that make it easier to reason about finite models. That is, for a world $\omega \in \Omega(\mathcal{R})$ and a quantified formula f over $\mathcal{R}$, $f(\omega)$ determinististally evaluates to T or F according to the semantics of f, which we will describe below. Critically, a quantified formula does not allow any relation to refer to an object twice (e.g. $(on\ x\ x)$). This is the one major difference between quantified formulas and standard first-order logic, and it critically effects the theorem provers and other reasoning algorithms.

Formally, a world ω is an $\mathcal{R}$-structure. It consists of a finite object set O and an interpretation I, which assigns each possible relation R over O to T or F.

We will now define all these concepts precisely.

Definition A *relation template* R is a pair $(name, arity)$ where $name \in Text$ is a text string and $arity \in \mathbf{N}$ is a natural number

Definition A *model* $O \subset Text$ is a set of objects, also called terms. Each term is identified by a text string. A finite model is a finite set of terms.

Definition An *interpretation* $I_{R,O}$ for a model O and a relation template R with arity k is a function $I : \overbrace{O \times \ldots \times O}^{k} \rightarrow \{T, F\}$.

Definition An *$\mathcal{R}$-structure* is a tuple $(A, I_{R_1,O}, I_{R_2,O}, \ldots, I_{R_m,O})$ consisting of a finite model A and an interpretation $I_{R_i,O}$ for each relation template $R_i \in \mathcal{R}$.

A *world*, *state*, *environment*, and $\mathcal{R}$-structure are all equivalent concepts. From now on, we will think of a world as having only one interpretation $I : R_i \times O^{k_i}$, which maps a relation symbol and a tuple of terms to T or F. $I(R_i, x_1, \ldots, x_{ki}) = I_{R_i,O}(x_1, \ldots, x_{k_i})$.

Definition The *universe* $\Omega(\mathcal{R})$ for a relation template set $\mathcal{R}$ is the set of all worlds over $\mathcal{R}$. $\Omega(\mathcal{R}, O) \subset \Omega(\mathcal{R})$ is the set of worlds over $\mathcal{R}$ whose object set contains O.

For reference, the above definitions are all standard notation of finite model theory [39].

2.2 Quantified Formulas

Quantified formulas are the way we reason about sets of worlds. Quantified formulas are the basis for specifying problem specifications, primitive actions, and programs. The following section defines the syntax and semantics of quantified formulas precisely. The notation can be skimmed on a first reading.

Definition A *quantified formula* ϕ over a relation template set $\mathcal{R}$ and a model O is a function $\phi : \Omega \rightarrow \{T, F\}$, where Ω is the set of all worlds $\omega(O')$ over $\mathcal{R}$ whose model $O' \supseteq O$ contains O.

Now, a language will be defined for quantified formulas, so that a formula can be easily evaluated on any given world ω. First, the syntax is defined:

$$\begin{aligned}
QF &::= Conj \mid Disj \mid Quantified \mid Expression \mid T \mid F \\
Conj &::= QF \wedge QF \wedge \ldots \wedge QF \\
Disj &::= QF \vee QF \vee \ldots \vee QF \\
Quantified &::= Quantifier\ Term\ QF \\
Quantifier &::= AtLeast(Quantity) \mid AtMost(Quantity) \\
Quantity &::= Nat \mid AllBut(Nat) \\
Expression &::= Relation \mid \neg Relation \\
Relation &::= (Name\ Term\ \ldots\ Term)
\end{aligned}$$

That is, a quantified formula is a conjunction of sub-formulas, a disjunction of sub-formulas, a quantification, a primitive expression, trivially true, or trivially false. A quantified formula can be thought of as a tree with primitive expressions or trivial booleans at the leaves. A primitive expression is either a relation or the negation of a relation. A quantification is a generalization of the $\forall$ and $\exists$ quantifiers of first-order logic. Nat is the natural numbers. With the extended notation, it is possible to specify that there exist more than one of a particular term and that there are at most a certain number of terms satisfying a particular sub-formula.

Before the semantics of sentences in the above grammar can be formally defined, we must first define the relation template set $\mathcal{R}$ and model O over which it works. The relation template set of a quantified formula ϕ is the set of all relations used in all of the leaves of the formula.

$$Rels(\phi) \quad = \quad \textbf{case } \phi \textbf{ of} \begin{cases} \phi_1 \wedge \ldots \wedge \phi_k & \rightarrow \bigcup_i Rels(\phi_i) \\ \phi_1 \vee \ldots \vee \phi_k & \rightarrow \bigcup_i Rels(\phi_i) \\ Q\ x\ \tilde{\phi} & \rightarrow Rels(\tilde{\phi}) \\ (Name\ x_1 \ldots\ x_k) & \rightarrow (Name, k) \\ \neg(Name\ x_1 \ldots x_k) & \rightarrow (Name, k) \\ T, F & \rightarrow \{\} \end{cases}$$

The free variables of a formula ϕ are defined to be the variables of leaf relations that are not quantified variables of an outer quantification.

$$Frees(\phi) \quad = \quad \textbf{case } \phi \textbf{ of} \begin{cases} \phi_1 \wedge \ldots \wedge \phi_k & \rightarrow \bigcup_i Frees(\phi_i) \\ \phi_1 \vee \ldots \vee \phi_k & \rightarrow \bigcup_i Frees(\phi_i) \\ Q\ x\ \tilde{\phi} & \rightarrow Frees(\tilde{\phi}) - \{x\} \\ (Name\ x_1 \ldots x_k) & \rightarrow \{x_1, \ldots, x_k\} \\ \neg(Name\ x_1 \ldots x_k) & \rightarrow \{x_1, \ldots, x_k\} \\ T, F & \rightarrow \{\} \end{cases}$$

The substitution notation is as follows: $\phi' = \phi[x/x']$ for $x \in Frees(\phi)$ means that ϕ' is produced from ϕ by replacing x with x' in all relations where x occurs free in ϕ. x' must be a fresh variable, i.e. not used as a variable in any quantification of ϕ.

Now the semantics of a quantified formula ϕ can be defined over a world $\omega = (O, I)$, which must be defined over the relation templates $Rels(\phi)$ and which must contain the objects $Frees(\phi)$. $\phi(\omega)$ is always defined with respect to an object set $A \subset O$, where A is the subset of objects that can be substituted into a quantification.

$$\phi(\omega)\,\|A\| = \textbf{case } \phi \textbf{ of } \begin{cases} \phi_1 \wedge \ldots \wedge \phi_k & \rightarrow \bigwedge_i \phi_i(\omega)\|A\| \\ \phi_1 \vee \ldots \vee \phi_k & \rightarrow \bigvee_i \phi_i(\omega)\|A\| \\ AtMost(n)\ x\ \tilde{\phi} & \rightarrow \bigwedge_{A' \in Choose(A,n+1)} \\ & \bigvee_{x' \in A'} \neg\tilde{\phi}(\omega)[x/x']\|A - \{x'\}\| \\ AtMost(AllBut(n))\ x\ \tilde{\phi} & \rightarrow \bigwedge_{A' \in Choose(A,|A|-n+1)} \\ & \bigvee_{x' \in A'} \neg\tilde{\phi}(\omega)[x/x']\|A - \{x'\}\| \\ AtLeast(n)\ x\ \tilde{\phi} & \rightarrow \bigvee_{A' \in Choose(A,n)} \\ & \bigwedge_{x' \in A'} \tilde{\phi}(\omega)[x/x']\|A - \{x'\}\| \\ AtLeast(AllBut(n))\ x\ \tilde{\phi} & \rightarrow \bigvee_{A' \in Choose(A,max(0,|A|-n))} \\ & \bigwedge_{x' \in A'} \tilde{\phi}(\omega)[x/x']\|A - \{x'\}\| \\ (Name\ x_1 \ldots x_k) & \rightarrow I(Name,\ x_1, \ldots, x_k) \\ \neg(Name\ x_1 \ldots x_k) & \rightarrow \neg I(Name, x_1, \ldots, x_k) \\ T & \rightarrow T \\ F & \rightarrow F \end{cases}$$

$Choose(A, n)$ is defined as $\{A' \in Pow(A).|A'| = n\}$. It is the subsets of A of size n. For the top-level formula, we define:

$$\phi(\omega) = \phi(\omega)\|O - Frees(\phi)\|$$

$AtMost(n)$ $\tilde{\phi}(x)$ and $AtLeast(n)$ $\tilde{\phi}(x)$ have a meaning that should be familiar. $AtMost(1)$ x $\tilde{\phi}(x)$ is equivalent to $Unique$ x $\tilde{\phi}(x)$ and $AtLeast(1)$ x $\tilde{\phi}(x)$ is equivalent to $\exists x$ $\tilde{\phi}(x)$ for example. Uniqueness does not imply existance. It means there is zero or one x satisfying $\tilde{\phi}(x)$. The $AllBut(n)$ quantities define maxima and minima relative to the size of the set A. For example, *AllBut*(0) refers to the entire set. So, the $\forall x\ \tilde{\phi}(x)$ quantifier is defined as $\mathit{AtLeast}(\mathit{AllBut}(0))\ x\ \tilde{\phi}(x)$. From now on, the $\forall$, $\exists$, U, and N(None) quantifiers will have this syntactic sugar.

$$\begin{aligned}
\forall x\ \tilde{\phi}(x) &\Leftrightarrow \mathit{AtLeast}(\mathit{AllBut}(0))\ x\ \tilde{\phi}(x) \\
\exists x\ \tilde{\phi}(x) &\Leftrightarrow \mathit{AtLeast}(1)\ x\ \tilde{\phi}(x) \\
Ux\ \tilde{\phi}(x) &\Leftrightarrow \mathit{AtMost}(1)\ x\ \tilde{\phi}(x) \\
Nx\ \tilde{\phi}(x) &\Leftrightarrow \mathit{AtMost}(0)\ x\ \tilde{\phi}(x)
\end{aligned}$$

What is unique about quantified formula semantics relative to first-order logic is the set A in the above formula, which specifies the objects of a world that are valid substitutes for the variables of a quantification. This construct allows makes it natural to express rules about relations between different objects without worrying about equality. Consider the formula $\forall x \forall y\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x)$. This means that "For all x, for all y *not equal* to x, x on y implies that y is not on x." When calculating quantified formula semantics, any two terms in a leaf relation will be distinct from each other. This fits into the intuitive model of worlds as graphs of objects and relations without the need to worry about odd special cases of self-directed edges (e.g. $(\mathit{on}\ x\ x)$). Under these semantics, any self-edge is meaningless.

Quantified formulas are not a new type of logic. Any quantified formula can be translated into a sentence of first-order logic with equality. However, algorithmically, quantified formulas are much easier to reason with and automatically prove theorems on. The algorithms that have been developed for SSGP depend heavily on these semantics. One immediate consequence of this choice is that inference using resolution and unification is difficult if not impossible. Free variables cannot be unified with anything, because of the possibility that two free variables in the same relation will become equal. Hence, the theorem provers and simplifiers of SSGP do not use resolution or unification to make inferences.

2.2.1 Syntactic Sugar

This section will introduce some further notation for dealing with quantified formulas. First, it will be shown that the negation of a quantified formula exists and is easy to compute.

Theoretically, the negation $\neg\phi$ of a quantified formula ϕ is the function on $\omega \in \Omega(\mathit{Rels}(\phi), \mathit{Frees}(\phi))$ such that $(\neg\phi)(\omega) = \neg(\phi(\omega))$. Operationally,

$$\neg\phi = \textbf{case } \phi \textbf{ of } \begin{cases} \phi_1 \wedge \ldots \wedge \phi_k & \rightarrow \neg\phi_1 \vee \ldots \vee \neg\phi_k \\ \phi_1 \vee \ldots \vee \phi_k & \rightarrow \neg\phi_1 \wedge \ldots \wedge \neg\phi_k \\ \mathit{AtLeast}(n)\ x\ \tilde{\phi} & \rightarrow \mathit{AtMost}(n-1)\ x\ \tilde{\phi} \\ \mathit{AtLeast}(\mathit{AllBut}(n))\ x\ \tilde{\phi} & \rightarrow \mathit{AtMost}(\mathit{AllBut}(n+1))\ x\ \tilde{\phi} \\ \mathit{AtMost}(n)\ x\ \tilde{\phi} & \rightarrow \mathit{AtLeast}(n+1)\ x\ \tilde{\phi} \\ \mathit{AtMost}(\mathit{AllBut}(n))\ x\ \tilde{\phi} & \rightarrow \mathit{AtLeast}(\mathit{AllBut}(n-1))\ x\ \tilde{\phi} \\ (\mathit{Name}\ x_1\ \ldots\ x_k) & \rightarrow \neg(\mathit{Name}\ x_1\ \ldots\ x_k) \\ \neg(\mathit{Name}\ x_1\ \ldots\ x_k) & \rightarrow (\mathit{Name}\ x_1\ \ldots\ x_k) \\ T & \rightarrow F \\ F & \rightarrow T \end{cases}$$

Using formula negation, it is possible to remove all *AtMost* quantifiers from a formula while maintaining equivalence. This is due to the following equation.

$$\begin{aligned} \mathit{AtMost}(n)\ x\ \tilde{\phi} &= \mathit{AtLeast}(\mathit{AllBut}(n))\ x\ \neg\tilde{\phi} \\ \mathit{AtMost}(\mathit{AllBut}(n))\ x\ \tilde{\phi} &= \mathit{AtLeast}(n)\ x\ \neg\tilde{\phi} \end{aligned}$$

Therefore, in future algorithms, it will be assumed that the quantified formulas have only *AtLeast* quantifiers and no *AtMost* quantifiers.

Two other standard logical operators, $\Rightarrow$(implication) and $\Leftrightarrow$ (equivalence) are defined in the standard way.

$$\begin{aligned}\phi_1 \Rightarrow \phi_2 &= \neg\phi_1 \vee \phi_2 \\ \phi_1 \Leftrightarrow \phi_2 &= (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)\end{aligned}$$

For concreteness, consider one example, the specification for a linked list:

$$\begin{aligned} & \forall x U y\ (\mathit{precedes}\ x\ y) \\ \wedge\ & \forall x \forall y\ (\mathit{connected}\ x\ y) \Leftrightarrow (\mathit{precedes}\ x\ y) \vee \\ & \qquad (\exists z\ (\mathit{connected}\ x\ z) \wedge (\mathit{connected}\ z\ y)) \\ \wedge\ & \forall x \forall y\ (\mathit{connected}\ x\ y) \Leftrightarrow \neg(\mathit{connected}\ y\ x)\end{aligned}$$

precedes means that there is a direct link between object x and object y. *connected* means that there is a path of links from x to y. The first sub-formula means that any object x can only link to at most one other object y. The second sub-formula defines *connected* to mean a path between x and y. The third sub-formula establishes that paths are anti-symmetric. There is always a path between x and y or between y and x, but there is only a path in one direction. There are many other ways to logically define a list, but this is the one shall be used throughout this thesis.

2.2.2 Subsumption and Simplification

These $\Rightarrow$ and $\Leftrightarrow$ operators introduce the important concepts of subsumption and simplification.

Definition Given quantified formulas ϕ_1 and ϕ_2, ϕ_1 ***subsumes*** ϕ_2 iff for all $\omega \in \Omega$, $\phi_1(\omega) \Rightarrow \phi_2(\omega)$. This is also true iff $\phi_1 \Rightarrow \phi_2$

Definition For quantified formulas ϕ_1 and ϕ_2, ϕ_1 is *equivalent* to ϕ_2 iff for all $\omega \in \Omega$, $\phi_1(\omega) \Leftrightarrow \phi_2(\omega)$. This is also true iff $\phi_1 \Leftrightarrow \phi_2$.

Definition The *size* or *complexity* of a quantified formula is the number of relation nodes in the formula tree plus the total number of quantifiers.

Definition For formulas ϕ_1 and ϕ_2, ϕ_1 is a *simplification* of ϕ_2 iff $\phi_1 \Leftrightarrow \phi_2$ and $size(\phi_1) < size(\phi_2)$.

Definition For a given formula G and two formulas ϕ_1 and ϕ_2, we say that $\phi_1 \Leftrightarrow_G \phi_2$ iff $\phi_1 \wedge G \Leftrightarrow \phi_2 \wedge G$. We say ϕ_1 is equivalent to ϕ_2 given G. If $size(\phi_2) < size(\phi_1)$, ϕ_2 is a simplification of ϕ_1 given G.

If $G \Rightarrow \phi_1$ then $\phi_1 \Leftrightarrow_G T$ and ϕ_1 simplifies to T given G. Many of the algorithms have a complexity that is dependent on the size of a quantified formula ϕ. Thus, it is important to have procedures for testing subsumption and for performing simplification. In general, testing if ϕ_1 subsumes ϕ_2 is a problem in first-order logic theorem proving and is undecidable in general. Nevertheless, for the examples encountered in practice, many theorem proving algorithms are practical. A fast and deep prover and simplifier are critical bits of infrastructure in the SSGP systems.

2.3 Sets, Properties, and Partitions

Definition A *property* is a relation with only one argument.

Another way to think about a property P in a world $\omega(O, I)$ is as the subset $A = \{x \in O.(P\ x)\}$, the elements of O under which P is true in interpretation I. Sometimes it is useful to think about properties as sets and vice versa. In particular, we often want to write quantified formulas that are quantified over a particular set A. The following equations define quantified formulas over sets. (Here, the same notation A is used for the set and its property.)

$$\begin{aligned} AtLeast(n)\ x{:}A\ \phi(x) &= AtLeast(n)\ x\ (A\ x) \wedge \phi(x) \\ AtLeast(AllBut(n))\ x{:}A\ \phi(x) &= AtLeast(AllBut(n))\ x\ \neg(A\ x) \vee \phi(x) \end{aligned}$$

Definition A *partition* on a world $\omega(O)$ is a group of sets, $A_1, \ldots, A_k$ such that $A_1 \cup \ldots \cup A_k = O$ and $A_i \cap A_j = \emptyset$ for $i \neq j$.

Partitions are very important for analyzing the roles of a world and finding recursive structure. In analysis, there will often be quantified formulas over unions of disjoint sets, and an algorithm will want to use facts it knows about the individual sets themselves. The following equations make this possible.

$$AtLeast(n)\ x{:}A_1 \cup \ldots \cup A_k\ \phi(x) = \bigvee_{n_1+\ldots+n_k=n} \bigwedge_i AtLeast(n_i)\ x{:}A_i\ \phi(x) \tag{2.1}$$

$$AtLeast(AllBut(n))\ x{:}A_1 \cup \ldots \cup A_k\ \phi(x) = \bigvee_{n_1+\ldots+n_k=n} \bigwedge_i AtLeast(AllBut(n_i))\ x{:}A_i\ \phi(x) \tag{2.2}$$

Definition A quantified formula is *normalized* if all of the sets in its quanitifications intersect at most one partition element. Equations 2.1 and 2.2 show how to convert an unnormalized formula into a normalized one.

The following example shows how this works for existence and uniqueness:

$$\begin{aligned}\exists x{:}A_1{\cup}A_2\ \phi(x) &= (\exists x{:}A_1\ \phi(x)) \vee (\exists x{:}A_2\ \phi(x))\\ Ux{:}A_1{\cup}A_2\ \phi(x) &= ((Nx{:}A_1\ \phi(x)) \wedge (Ux{:}A_2\ \phi(x)))\ \vee\\ &\quad ((Ux{:}A_1\ \phi(x)) \wedge (Nx{:}A_2\ \phi(x)))\end{aligned}$$

Throughout the following sections, we will often talk about quantifying a formula ϕ over a partition $\{T_1, \ldots, T_k\}$. This means finding all quantifications $Q\ x\ \phi(x)$, changing them to $Q\ x{:}\{T_1 \cup \ldots \cup T_k\}\ \phi(x)$ and then normalizing.

Frequently, we have to deal with quantified formulas that are quantified over multiple overlapping partitions $\{S_i\}$ and $\{T_j\}$. In this case, the normalized quantifications have the form $Q\ x{:}S_i \cap T_j\ \phi(x)$.

A special kind of partition and normalization occurs when we need to isolate an individual object a out of a formula. In this case, the partition is $\{\{a\}, O-\{a\}\}$. This is called *breaking out* an element. The following two equations show how breaking out is done with existential, universal, and uniqueness quantifiers. Note that the logic of quantified formulas makes this easy to do.

$$\begin{array}{rcl} \exists x\ \phi(x) & \rightarrow & \phi(a)\ \vee \exists x\ \phi(x) \\ \forall x\ \phi(x) & \rightarrow & \phi(a)\ \wedge\ \forall x\ \phi(x) \\ Ux\ \phi(x) & \rightarrow & \quad (\neg\phi(a)\ \wedge\ Ux\ \phi(x)) \\ & & \vee\ \ (\phi(a)\ \wedge\ Nx\ \phi(x)) \end{array}$$

If a quantified formula is already quantified over one or more partitions $\{T\}$, to break out an element a, one must know which sets of the partitions it belongs to. For example, if one has the formula $\phi = \forall x{:}A\ \exists y{:}A\ (\mathit{connected}\ x\ y) \vee \exists y{:}B\ (\mathit{connected}\ x\ y)$, one has to know whether a belongs to A or B. If $a \in A$, then the breakout becomes:

$$\begin{array}{l} \quad \forall x{:}A\ ((\mathit{connected}\ x\ a)\ \vee\ \exists y{:}A\ (\mathit{connected}\ x\ y))\ \vee\ \exists y{:}B\ (\mathit{connected}\ x\ y) \\ \wedge\ \ (\exists y{:}A\ (\mathit{connected}\ a\ y)\ \vee\ \exists y{:}B\ (\mathit{connected}\ x\ y)) \end{array}$$

If $a \in B$, the breakout becomes:

$$\forall x{:}A\ \exists y{:}A\ (\mathit{connected}\ x\ y)\ \vee\ (\mathit{connected}\ x\ a)\ \vee\ \exists y{:}B\ (\mathit{connected}\ x\ y)$$

2.4 Size Possibilities

An interesting class of quantified formulas are ones where a trivial formula (T or F) appears inside a quantification. Consider the formula $\mathit{AtLeast}(n)\ x{:}A\ T$. If $|A|$, the size of A, is at least n, than this statement is true(T). If $|A| < n$, then it is false(F). This is equivalent to stating that the set A has at least n elements. A short-hand notation for this is $(\mathit{AtLeast}(n)\ A)$. Another such formula is $\mathit{AtLeast}(\mathit{AllBut}(n))\ x$:

A F. If $|S| > n$, than this statement is equivalent to F. If $|A| \leq n$, than it is T. So the formula means that A has at most n elements. It is written $(AtMost(n)\ A)$. Thus, these types of quantified formulas can be used to express any kind of size constraints on individual sets. The following table summarizes this and defines a few common size constraints that will be used.

$$\begin{aligned}
\mathit{AtLeast}(n)\ x{:}A\ T &= (\mathit{AtLeast}(n)\ \ A) \\
\mathit{Atleast}(\mathit{AllBut}(n))\ x{:}A\ T &= T \\
\mathit{AtLeast}(n)\ x{:}A\ F &= \textbf{if } n = 0 \textbf{ then } T \textbf{ else } F \\
\mathit{AtLeast}(\mathit{AllBut}(n))\ x{:}A\ F &= (\mathit{AtMost}(n)\ A) \\
(\mathit{AtMost}(0)\ A) &= (\mathit{None}\ A) \\
(\mathit{AtLeast}(1)\ A) &= (\mathit{Exists}\ A) \\
(\mathit{AtMost}(1)\ A) &= (\mathit{Unique}\ A) \\
(\mathit{Exists}\ A) \wedge (\mathit{Unique}\ A) &= (\mathit{Defined}\ A)
\end{aligned}$$

When the quantifier is over the entire state (i.e. not just a set A), we will write (Exists) or $(\mathit{Defined})$ (meaning the state has at least one element or the state has exactly one element).

2.5 Problem Specifications

Now that there is a common notation and semantics for worlds and quantified formulas, program specifications can be specified precisely for the SSGP systems to solve. The output program is unique to each system, so the description of the representation and semantics of the output programs is deferred for each chapter.

Form2Grammar takes a single quantified formula S. Its purpose is to produce a recursive grammar that characterizes all worlds that satisfy S. This grammar can then be used to prove properties of worlds satisfying S by structural induction.

Spec2Action takes two quantified formulas, S and G. S, the start formula, is a quantified formula describing all the worlds that could be presented as input to the action program. It is like a precondition. G is the goal formula or postcondition. The purpose of Spec2Action is to find an action template for manipulating a subset of the relations of the world. For any world satisfying S at the beginning, after the manipulations have been performed, the new world will satisfy G. Spec2Action uses the structure of Form2Grammar to prove by structural induction that its manipulations can always be done.

HELPS also takes two quantified formulas, S and G. However, it cannot manipulate individual relations freely and independently. It can only manipulate the relations through a set of primitive actions $\mathcal{ACT}$. Chapter 5 will describe the representation for primitive actions, as well as the programs to solve them. These structures all use quantified formulas as the basic components.

Chapter 3

Form2Grammar

Form2Grammar is the first SSGP system that will be examined. It takes a quantified formula and finds a recursive structure that all worlds satisfying the formula must decompose to. Previous program synthesis and general planning systems have taken the recursive structure of their input domains as a given. This is a somewhat unrealistic assumption for many autonomous agents, and so it is important that an agent be able to discover recursive structure from a purely logical formula.

This chapter begins with the representation for a grammar, and then it shows an example of an input formula for a marked list with a correct output grammar. The next section describes the example problems that have been used to evaluate the system. Section 3 presents the complete architecture for the system with demonstrations from the marked list example. The formal notation and definitions for all of the concepts can be difficult to understand on a first reading. A recommended way to read the chapter is to focus on the provided examples and skim the explanations with full generality.

3.1 Grammar Representations

The overall format of a grammar is similar to the Backus-Naur grammars that are standard in compilers and natural language processing. It consists of non-terminals which are either disjunctions of other non-terminals, *decompositions*, or ϵ, the empty world.

$$
\begin{aligned}
NT &::= NT_1|NT_2|\ldots|NT_k \\
NT &::= Piv; \{\{A_{11},\ldots,A_{1k_1}\} \rightarrow NT_1,\ldots,\{A_{m1},\ldots,A_{mk_m}\} \rightarrow NT_m\}; \chi \\
NT &::= \epsilon
\end{aligned}
$$

A grammar is defined over a relation template set $\mathcal{R}$. Each non-terminal NT defines the set of worlds $\omega \in \Omega(\mathcal{R})$ that it produces. In turn, each grammar has a top non-terminal, Top, which defines the worlds ω that are produced by the grammar as a whole. A decomposition is the most interesting non-terminal, because it defines how larger worlds are produced from smaller ones. In a decomposition, there is a pivot point Piv and one or more sub-productions, $NT_1, \ldots, NT_m$. Each sub-production NT_i in turn partitions into k_i subsets, $\{A_{i1},\ldots,A_{ik_i}\}$. The last component of a subproduction is its *composing interpretation* or cross relation set χ. This is a function that assigns T or F to all relations between the sub-productions and the pivot and all relations between different sub-productions. The subsets A_{ij} let us compactly represent χ since all relations between A_{ij} and the pivot have the same truth value, and all relations between A_{ij} and A_{kl} of different sub-productions ($i \neq k$) will have the same truth value. That is, for a binary relation R, $\chi((R\ Piv\ A_{ij}))$ and $\chi((R\ A_{ij}\ Piv))$ are well-defined, as are $\chi((R\ A_{ij}\ A_{kl}))$ and $\chi((R\ A_{kl}\ A_{ij}))$ when $i \neq k$. The composing interpretation is only defined between sub-productions and between sub-productions and the pivot. It is not defined within the sub-productions. The interpretation of relations within a sub-production comes entirely from that sub-production itself.

For clarity, consider the simple example of a grammar for a linked list. The logical definition of a list, already defined in Chapter 2, is

$$
\begin{aligned}
& \forall x U y\ (\textit{precedes}\ x\ y) \\
\wedge\ & \forall x \forall y\ (\textit{connected}\ x\ y) \Leftrightarrow (\textit{precedes}\ x\ y) \vee \\
& \qquad (\exists z\ (\textit{connected}\ x\ z) \wedge (\textit{connected}\ z\ y)) \\
\wedge\ & \forall x \forall y\ (\textit{connected}\ x\ y) \Leftrightarrow \neg(\textit{connected}\ y\ x)
\end{aligned}
$$

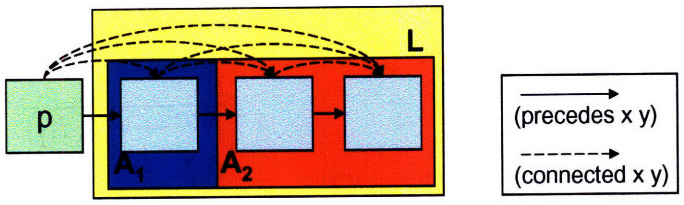

Figure 3-1: One breakdown or decomposition of a grammar for a singly linked list

One grammar for the list is:

$$
\begin{aligned}
L &::= P|\epsilon \\
P &::= p; \{\{A_1, A_2\} \rightarrow L\}; \{(\mathit{precedes}\ p\ A_1), \neg(\mathit{precedes}\ A_1\ p), (\mathit{connected}\ p\ A_1), \\
&\quad \neg(\mathit{connected}\ A_1\ p), \neg(\mathit{precedes}\ p\ A_2), \neg(\mathit{precedes}\ A_2\ p), (\mathit{connected}\ p\ A_2), \\
&\quad \neg(\mathit{connected}\ A_2\ p)\}
\end{aligned}
$$

A list has two relations ($\mathit{precedes}$?1 ?2) and ($\mathit{connected}$?1 ?2) defining a link and a path of links respectively. The grammar specifies a list as (None) or a decomposition. In the decomposition, the pivot is the first element of the list and there is one sub-production, which is the rest of the list. The rest sub-list partitions into two sub-sets, the first of which links directly to the head and the second of which is only connected to the head. An illustration of the list break-down, p, A_1, A_2 for a list of four elements is shown in Figure 1.

A list has only one sub-production, which is the rest of the list. An example structure with two sub-productions would be a tree, which has one sub-production for the left branch and one sub-production for the right branch.

To be precise, the semantics of a grammar can be defined. This will be done via production rules (i.e. what larger worlds (O, I) can be produced from smaller worlds $\{(O_i, I_i)\}$). The three production rules are:

$$\begin{array}{lcc}
NT ::= & \epsilon & \dfrac{}{NT \hookrightarrow (\{\}, I_{\{\}})} \\
NT ::= & NT_1 | \ldots | NT_k & \dfrac{NT_i \hookrightarrow (O_i, I_i)}{NT \hookrightarrow (O_i, I_i)} \\
NT ::= & Piv; \{\{A_{ij}\} \to NT_i\}; \chi &
\end{array}$$

$$\frac{NT_1 \hookrightarrow (O_1, I_1) \ \ldots \ NT_k \hookrightarrow (O_k, I_k) \ \ldots \ O_i = Q_{i1} \cup \ldots \cup Q_{in_i} \ldots}{NT \hookrightarrow (\{Piv\} \cup O_1 \cup \ldots \cup O_k, Lift(\chi, \{\{Q_{ij}\}\}, I_1, \ldots, I_k))}$$

$Lift$ is a function to produce an interpretation on $\{Piv\} \cup \bigcup_i O_i$ from the smaller interpretations $I_1, \ldots, I_k$, the composing interpretation, and an arbitrary set of partitions on the sets $O_1, \ldots, O_k$. Let $L = Lift(\chi, \{\{Q_{ij}\}\}, I_1, \ldots, I_k)$. Let $R = (Name\ T_1 \ldots T_n)$. If $\{T_j\} \subset O_i$, then $L(R) = I_i(R)$. That is, the relation is entirely within one sub-world (O_i, I_i), so L defers to that world's interpretation. Otherwise, let $T_j \in Q_{i_j l_j}$. To count the pivot, let $Q_{00} = A_{00} = \{Piv\}$. In this case, $L(R) = \chi((Name\ A_{i_1 l_1} \ldots\ A_{n_1 l_1}))$. The lifted interpretation value comes from the cross-relation evaluated with the composition interpretation χ.

Note that the partition $\{\{Q_{ij}\}\}$ of the sub-productions is arbitrary, and it may have nothing to do with the decomposition of the sub-productions themselves. This is a problem for inductive reasoning. And, the sets of worlds ω that are produced by a grammar of this kind is too big. Thus, we augment the grammar so that each non-terminal NT_i is also associated with a constraining quantified formula ϕ_i. Any world that does not satisfy $\phi_i$is not produced by NT_i. We call such a grammar a *constraining grammar.* We can easily rewrite the grammar productions above to incorporate formulas.

$$\begin{array}{lcc}
(NT, \phi) ::= & \epsilon & \dfrac{\phi(\{\}, I_{\{\}})}{NT \hookrightarrow (\{\}, I_{\{\}})} \\
(NT, \phi) ::= & NT_1 | \ldots | NT_k & \dfrac{NT_i \hookrightarrow (O_i, I_i)\ \phi(O_i, I_i)}{NT \hookrightarrow (O_i, I_i)} \\
(NT, \phi) ::= & Piv; \{\{A_{ij}\} \to NT_i\}; \chi &
\end{array}$$

$$\frac{...NT_i \hookrightarrow (O_i, I_i) \quad \ldots \; O_i = Q_{i1} \cup \ldots \cup Q_{in_i} \ldots \;\; \phi(O, L)}{NT \hookrightarrow (O = \{Piv\} \cup O_1 \cup \ldots \cup O_k, L = Lift(\chi, \{\{Q_{ij}\}\}, I_1, \ldots, I_k))}$$

Now we can formally state the goal of Form2Grammar. Given a quantified formula S, find a constraining grammar that produces exactly those worlds ω that S satisfies. That is, find $Gram = \{(Top, S), (NT_1, \phi_1), \ldots, (NT_k, \phi_k)\}$ so that $(Top \hookrightarrow (O, I))$ if and only if $S(O, I) = T$. All of the example problems studied have had relations of arity at most 2. Thus, the Form2Grammar system only works on worlds with such relations. Furthermore, Form2Grammar only works on quantified formulas without free variables. that is, every evariable in every formula must be quantified in some way.

3.2 Challenge Problems

The exercising problems for Form2Grammar are basic data structures: lists, trees, etc. Even though the ultimate purpose of the SSGP systems is not software program synthesis, data structures provide good coverage of interesting states with a lot of recursive structure. They best illustrate the difficulties and challenges that Form2Grammar must deal with. The structures considered are:

- singly linked list
- doubly linked list
- sorted singly linked list
- singly linked list with one marked element
- circularly linked list
- circularly linked list with one marked element
- binary search tree

$$\begin{array}{ll} & \forall x U y\ (\mathit{precedes}\ x\ y) \\ \wedge & \forall x \forall y\ (\mathit{connected}\ x\ y) \Leftrightarrow (\mathit{precedes}\ x\ y) \vee \\ & \quad (\exists z\ (\mathit{connected}\ x\ z) \wedge (\mathit{connected}\ z\ y)) \\ \wedge & \forall x \forall y\ (\mathit{connected}\ x\ y) \Leftrightarrow \neg(\mathit{connected}\ y\ x) \\ \wedge & Dx\ (\mathit{marked}\ x) \end{array} \tag{3.1}$$

Figure 3-2: Specification for a marked list

$$\begin{array}{rcl} L & ::= & M_1 \mid M_2 \\ M_1 & ::= & p;\ \{\{A_1, A_2\} \rightarrow L\};\ \neg(\mathit{marked}\ p),\ (\mathit{precedes}\ p\ A_1),\ (\mathit{connected}\ p\ A_1), \\ & & \quad (\mathit{connected}\ p\ A_2) \\ M_2 & ::= & p;\ \{\{A_1, A_2\} \rightarrow N\};\ (\mathit{marked}\ p),\ (\mathit{precedes}\ p\ A_1), \\ & & \quad (\mathit{connected}\ p\ A_1),\ (\mathit{connected}\ p\ A_2) \\ N & ::= & \epsilon | N' \\ N' & ::= & p;\ \{\{A_1, A_2\} \rightarrow N\};\ \neg(\mathit{marked}\ p),\ (\mathit{precedes}\ p\ A_1),\ (\mathit{connected}\ p\ A_1), \\ & & \quad (\mathit{connected}\ p\ A_2) \end{array} \tag{3.2}$$

Figure 3-3: Grammar for a marked list

- min heap
- BlocksWorld

These cases will be examined in detail in the results section including their full logical specifications. For illustration, consider an example of moderate complexity, a singly linked list with one marked element. The logical specification is shown in Figure 3-2.

D is the $Defined$ quantifier, meaning *Exists* and *Unique*. This is the same spec as the ordinary linked list (Equation 3.3) with one added condition: there is one element x which is *Marked*. This data structure is used to specify programs that will manipulate one element in a list, such as removing the element from the list or moving the element to the front.

A grammar for this shown in Figure 3-3

Wherever we have omitted some cross relations such as $\neg(\mathit{connected}\ A_2\ p)$, these relations are by default false.

$$\begin{array}{rcc}
\phi_L & = & listDef \wedge Dx\ (marked\ x) \\
\phi_N & = & listDef \wedge \forall x\ \neg(marked\ x) \\
\phi_{N'} & = & \phi_N : \{A_1, A_2\} \wedge \neg(marked\ p) \wedge \forall y : A_1(precedes\ p\ y) \\
 & \wedge & \forall y : A_1(connected\ p\ y) \wedge \forall y : A_2(connected\ p\ y) \\
 & \wedge & \forall y : A_1\neg(precedes\ y\ p) \wedge \forall y : A_1\neg(connected\ y\ p) \\
 & \wedge & \forall y : A_2\neg(precedes\ y\ p) \wedge \forall y : A_2\neg(connected\ p\ y) \\
 & \wedge & \forall y : A_2\neg(precedes\ p\ y) \\
\phi_{M_1} & = & \phi_L : \{A_1, A_2\} \wedge \neg(marked\ p) \wedge \forall y : A_1(precedes\ p\ y) \\
 & \wedge & \forall y : A_1(connected\ p\ y) \wedge \forall y : A_2(connected\ p\ y) \\
 & \wedge & \forall y : A_1\neg(precedes\ y\ p) \wedge \forall y : A_1\neg(connected\ y\ p) \\
 & \wedge & \forall y : A_2\neg(precedes\ y\ p) \wedge \forall y : A_2\neg(connected\ p\ x) \\
 & \wedge & \forall y : A_2\neg(precedes\ p\ y) \\
\phi_{M_2} & = & \phi_N : \{A_1, A_2\} \wedge (marked\ p) \wedge \forall y : A_1(precedes\ p\ y) \\
 & \wedge & \forall y : A_1(connected\ p\ A_1) \wedge \forall y : A_2(connected\ p\ y) \\
 & \wedge & \forall y : A_1\neg(precedes\ y\ p) \wedge \forall y : A_1\neg(connected\ y\ p) \\
 & \wedge & \forall y : A_2\neg(precedes\ y\ p) \wedge \forall y : A_2\neg(connected\ y\ p) \\
 & \wedge & \forall y : A_2\neg(precedes\ p\ y)
\end{array}$$

Figure 3-4: Constraining formulas for the marked list grammar

Let *listDef* be

$$\begin{aligned}
listDef = \quad & \forall x U y\ (precedes\ x\ y) \\
\wedge \quad & \forall x \forall y\ (connected\ x\ y) \Leftrightarrow (precedes\ x\ y) \vee \\
& (\exists z\ (connected\ x\ z) \wedge (connected\ z\ y)) \\
\wedge \quad & \forall x \forall y\ (connected\ x\ y) \Leftrightarrow \neg(connected\ y\ x)
\end{aligned} \tag{3.3}$$

The constraining quantified formulas associated with each non-terminal are in Figure 3-4.

$\phi_L : \{A_1, A_2\}$ and $\phi_N : \{A_1, A_2\}$ represent the formulas ϕ_L and ϕ_N respectively quantified over the disjoint set union $A_1 \cup A_2$. In the next section, we will see how such constraints are generated and why they appear as they do.

Each grammar production (disjunction, decomposition, and null) corresponds to a quantified formula. Thus, the productions, together with the constraining formulas, provide a set of proof obligations that the theorem prover can then tackle. The next section shows how grammars are produced inductively and then how their correctness

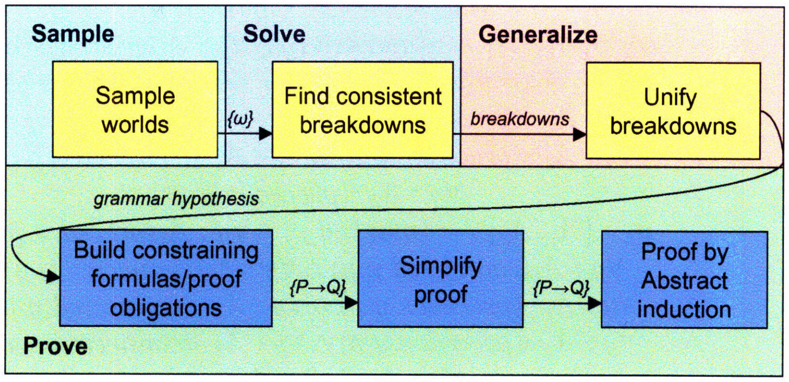

Figure 3-5: The architecture of Form2Grammar

is proved.

3.3 Form2Grammar Architecture

3.3.1 Overview

The components of Form2Grammar are shown in Figure 3.3.1. For the starting formula S, a set of examples is generated of various sizes. These examples are then broken down by a greedy algorithm that looks for pivot decompositions that are simple and consistent. The output is the most consistent decomposition among all of the examples. Then, the decompositions are unified to produce a hypothesis grammar. With the candidate grammar, a set of constraining formulas are produced to make a hypothesis constraining grammar.

Finally, Form2Grammar must show that the hypothesis constraining grammar is correct. Certain non-terminals must show that their constraining formulas deduce the formulas defined by their decompositions. So, each non-terminal generates a proof obligation. This proof proceeds in three stages. First, minimally sufficient conditions are found for the pivot point and the sub-production partition sets A_{ij}. This produces a smaller, simpler formula to prove. Second, the sets A_{ij} are replaced with properties, and the result formula is simplified further. Finally, the resulting formula is proven T

or there is a further proof to perform. This is done by the abstract induction theorem prover, a novel algorithm developed for the SSGP systems. This algorithm can prove theorems that need proof by induction, but it does not require any domain-specific induction rule.

Each component of the system will now be analyzed in turn. To manage the complexity of the algorithms great detail, all of the components will be demonstrated on the example of the marked linked list above.

3.3.2 Sampling instances

The goal of this component is to take a formula S and produce a random set of worlds $\omega_1, \ldots, \omega_k$ of various sizes that all satisfy S. First, all satisfying examples $\omega_1, \ldots, \omega_m$ with sizes between 0 and 3 are generated to get full coverage of all of the small cases. Then an additional N worlds are chosen $\omega_{m+1}, \ldots, \omega_{m+N}$ of random larger sizes. We assume that the desired number N is a provided system parameter (N is typically between 5 and 10). For each larger sample ω, a random number s is chosen uniformly between 1 and r, where r is another system parameter (typically $r = 5$). The sample size is then $size = s + 3$.

Now the problem is: given $size$, generate a random world ω of that size. The way to do this is to generate a propositional formula that is a concrete representation of S. Then, a SAT solver can be used to find one or all solutions to that formula. For example, consider the formula Dx (*marked* x). If $size = 3$, then Form2Grammar would first create objects $O = \{x_1, x_2, x_3\}$. Then, it would instantiate the formula:

$$\begin{array}{rl} & (\textit{marked } x_1) \vee (\textit{marked } x_2) \vee (\textit{marked } x_3) \\ \wedge & \neg(\textit{marked } x_1) \wedge \neg(\textit{marked } x_2) \\ \vee & \neg(\textit{marked } x_1) \wedge \neg(\textit{marked } x_3) \\ \vee & \neg(\textit{marked } x_2) \wedge \neg(\textit{marked } x_3) \end{array}$$

The SAT Solver would find three solutions.

$$\{\{(marked\ x_1)\ \neg(marked\ x_2)\ \neg(marked\ x_3)\},$$
$$\{\neg(marked\ x_1)\ (marked\ x_2)\ \neg(marked\ x_3)\},$$
$$\{\neg(marked\ x_1)\ \neg(marked\ x_2)\ (marked\ x_3)\}\}$$

All of these are equivalent to each other. Two worlds ω_1 and ω_2 are equivalent when ω_1 can be produced from ω_2 by just renaming the objects in the object set. As an aside, Form2Grammar has a simple utility to test whether two worlds are equivalent. Essentially, this is a problem of graph isomorphism. While there are no polynomial-time algorithms to do graph isomorphism, in practice the worlds that are compared can be equivalence-tested very quickly.

Generating a random world involves a slight modification to the SAT algorithm to search for a *random* satisfying assignment. Instead of choosing a proposition and the value to set it to ($True$ or $False$) according to some heuristic, the proposition's value is chosen randomly. The modified SAT algorithm is in Algorithm 1.

Random SAT is much slower than ordinary SAT, because ordinary SAT uses powerful heurstics to choose whether to set a proposition true or false. Therefore, once a proposition value has been set, ordinary SAT is run on the new assignment set to see if the assignment set can be made satisfiable and if the random SAT should continue down its path.

Let us proceed with the marked list example. The sample generator produces all six marked lists of sizes between 0 and 3 (6=3+2+1). Additionally, it produces 5 lists of sizes 7,8,5,6, and 8 respectively. The marked element occurs at positions 7,2,1,6, and 1 respectively. The next section shows how to hypothesize a grammar from these 11 examples.

Algorithm 1 Augmentation of SAT to return a random satisfying assignment

```
RandomSATSolve(PropositionalFormula, Assignments) {
  if (IsImpossible(PropositionalFormula, Assignments))  {
    return null;
  }
  if (AllPropositionsAssigned(PropositionalFormula,
                              Assignments)) {
    return Assignment;
  }
  Prop = ChooseUnassignedProposition(PropositionalFormula,
                                     Assignments);
  PropValue = ChooseRandom({True, False});
  Test = SATSolve(PropositionalFormula, Assignments +
                  {Prop=PropValue});
  if (Test != null)  {
    return RandomSATSolve(PropositionalFormula, Assignments +
                          {Prop=PropValue});
  }
  else {
    return RandomSATSolve(PropositionalFormula, Assignments +
                          {Prop=!PropValue});
  }
}
```

Algorithm 2 Decomposition Search

```
DecompositionSearch(Samples) {
  Nbest := {({},Samples)};
  while (HasDecompositions(Nbest)) {
    NextNbest := {};
    for (PartialDecomposition=(Pivots, SubSamples) in Nbest) {
      WorstSample = findWorstSample(PartialDecomposition);
      for (Term in Objects(WorstSample)) {
        (PivotDecomposition, SubWorlds) = Decompose(Term,
                                                    WorstSample);
        NextPartialDecomposition = (Pivots + PivotDecomposition,
            SubWorlds + SubSamples - {WorstSample});
        NextNbest := NextNbest + NextPartialDecomposition;
      }
    }
    Nbest := Top(N, NextNbest);
  }
  return Top(Nbest);
}
```

3.3.3 Example Decomposition

The strategy underlying inductive grammar synthesis is to find the simplest grammar that explains all of the examples. Thus, it looks for pivots that produce simple decompositions. Also, each decomposition yields smaller worlds that need to be further decomposed. So, the decomposition algorithm additionally searches for pivots that are consistent with previous decompositions. The search is an N-best beam search. At any point in the algorithm, there are N partial decompositions. Each partial decomposition contains all of the samples, partially broken down into pivots.

In one search iteration, for each partial decomposition, the worst sample sub-world is chosen, the one with the worst pivots. All possible pivots are evaluated on this worst world, producing a set of candidate next-level partial decompositions. Then, the best N next-level partial decompositions are chosen from among these new possibilities. The search continues until all examples have been fully decomposed. Then, the best decomposition from the final N-best list is returned.

The algorithm is summarized here:

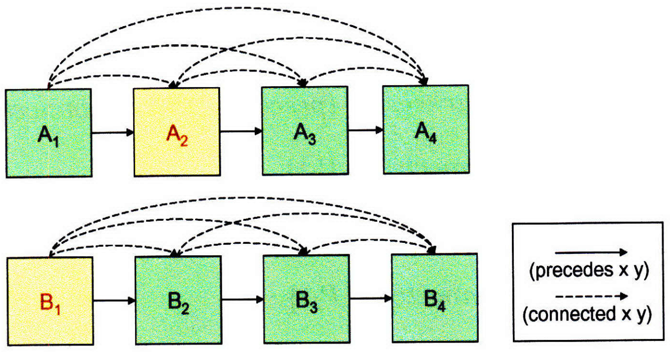

Figure 3-6: Two samples to break down

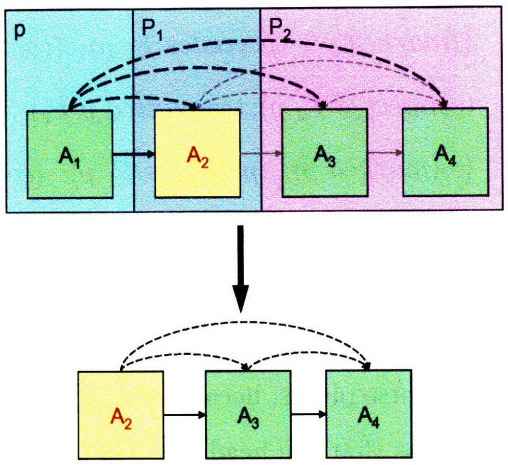

Figure 3-7: An example breakdown for the pivot $p = A_1$

Consider an example. Imagine that Form2Grammar starts with 2 samples of length 4 lists, shown in Figure 3-6.

Let N=2. An example decomposition with the pivot $p = A_1$ is shown in Figure 3-7.

At first, Form2Grammar considers every term as a potential pivot. In total, there are six possible pivot decompositions among the eight terms.

$$
\begin{aligned}
p = A_1 &\Rightarrow \{\neg(\textit{marked}\ p);\ (\textit{precedes}\ p\ P_1);\ (\textit{connected}\ p\ P_1);\\
&\quad (\textit{connected}\ p\ P_2)\}\\
p = B_1 &\Rightarrow \{(\textit{marked}\ p);\ (\textit{precedes}\ p\ P_1);\ (\textit{connected}\ p\ P_1);\\
&\quad (\textit{connected}\ p\ P_2)\}\\
p = A_4, p = B_4 &\Rightarrow \{\neg(\textit{marked}\ p);\ (\textit{precedes}\ P_1\ p);\ (\textit{connected}\ P_1\ p);\\
&\quad (\textit{connected}\ P_2\ p)\}\\
p = B_2 &\Rightarrow \{\neg(\textit{marked}\ p);\ (\textit{precedes}\ P_1\ p);\ (\textit{connected}\ P_1\ p);\\
&\quad (\textit{precedes}\ p\ P_2); (\textit{connected}\ p\ P_2);\ (\textit{connected}\ p\ P_3)\}\\
p = A_2 &\Rightarrow \{(\textit{marked}\ p);\ (\textit{precedes}\ P_1\ p);\ (\textit{connected}\ P_1\ p);\\
&\quad (\textit{connected}\ P_2\ p); (\textit{connected}\ p\ P_2);\ (\textit{connected}\ p\ P_3)\}\\
p = A_3, p = B_3 &\Rightarrow \{\neg(\textit{marked}\ p);\ (\textit{precedes}\ P_1\ p);\ (\textit{connected}\ P_1\ p);\\
&\quad (\textit{connected}\ P_2\ p); (\textit{precedes}\ p\ P_3);\ (\textit{connected}\ p\ P_3)\}
\end{aligned} \tag{3.4}
$$

When $p = A_1$, the decomposition sets P_1 and P_2 are $\{A_2\}$ and $\{A_3, A_4\}$. The interior terms are considered worse pivots, because they break the example down into 3 groups instead of the 2 groups that the head and tail terms do.

The worst sample is the one that has the worst best pivot. Since A and B both have 2-group pivots, both of them are considered worst. Then, the two head terms and the two tail terms all become next decompositions, and the next N-best list has 4 partial decompositions. The N-best list can have more than N partial decompositions if there are ties.

In the next round, consider the partial decomposition for $p = A_1$ 3.4. List A has a pivot, the second element a_2, which breaks down identically to the first pivot. List B does not have such a pivot. Therefore, B is the worst sample. Its two best pivots are b_1, the head of the list and b_4, the tail. After all level-1 partial decompositions have been considered, there are 4 level-2 partial decompositions that go to the next round.

At the end of the search, when all objects have been made into pivots in the candidate decompositions, there are 2 best decompositions: one that breaks down the lists from the front and one that breaks down the lists from the back. Both of these are equivalently simple, so the algorithm could return either one.

To summarize the search heuristics, pivot $x >$ pivot y if the breakdown induced by x has already been made before, or if x's breakdown is smaller than y's. Sample $X >$ sample Y if the best pivot in X is better than the best pivot in Y. If the best pivots are the same, X is better than Y if X is smaller than Y. Decomposition $P_1 >$ decomposition P_2 if P_1 has fewer sub-components than P_2. If the number of sub-components is the same, $P_1 > P_2$ if the pivots of P_1 are better than the pivots of P_2 starting with the worst pivots.

For the 11 marked list examples above, 11 decompositions are produced. The best decomposition is either the one that chooses pivots from the front of each sub-list, or the one that chooses pivots from the rear. These are the two decomposition schemes in Figure 3-8. For the rest of this section, we will work with the decomposition that starts from the front.

3.3.4 Decomposition Unification

Once a candidate decomposition has been made for the examples, the decompositions must be unified to produce a single grammar. This is the step of Form2Grammar that requires the largest inductive leap. There are potentially many different grammars that could all produce the decompositions seen in the examples. We want to choose a grammar that is simple and generic, but we don't want to pick a grammar that is too general and would admit too many worlds.

The algorithm consists of first building a finite-state machine to represent all of the examples. The nodes of this machine are the sub-instances of the world decompositions. The arcs connecting the nodes are labeled by the type of pivot breakdown they represent. If a world O breaks into more than one subset, by breakdown $b = (piv;\ A_1, \ldots, A_k)$ e.g., each A_i is a separate node and the link from O to A_i is labeled (b, i).

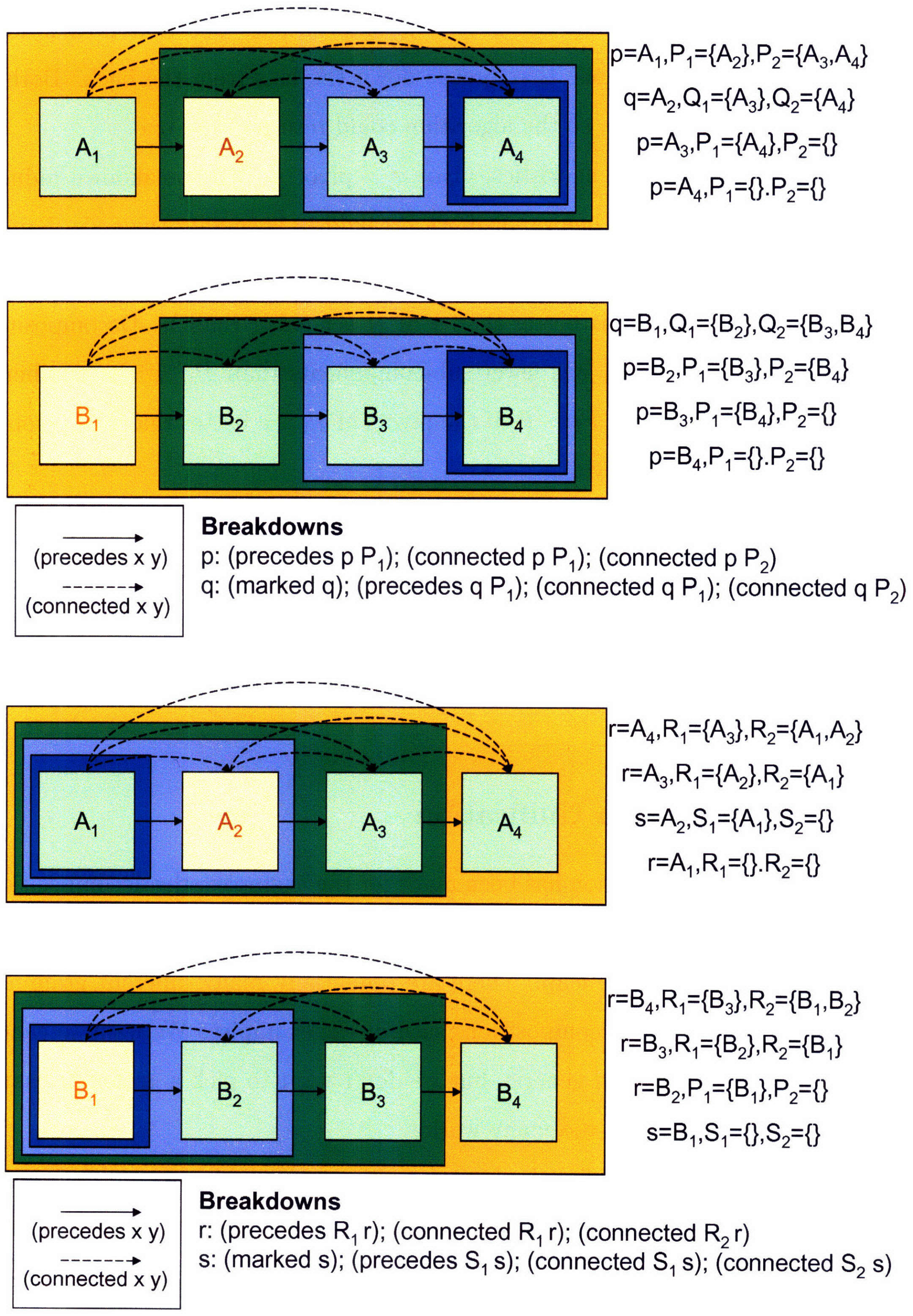

Figure 3-8: The two simplest sample break-downs. The first takes pivots from the front. The second takes pivots from the rear.

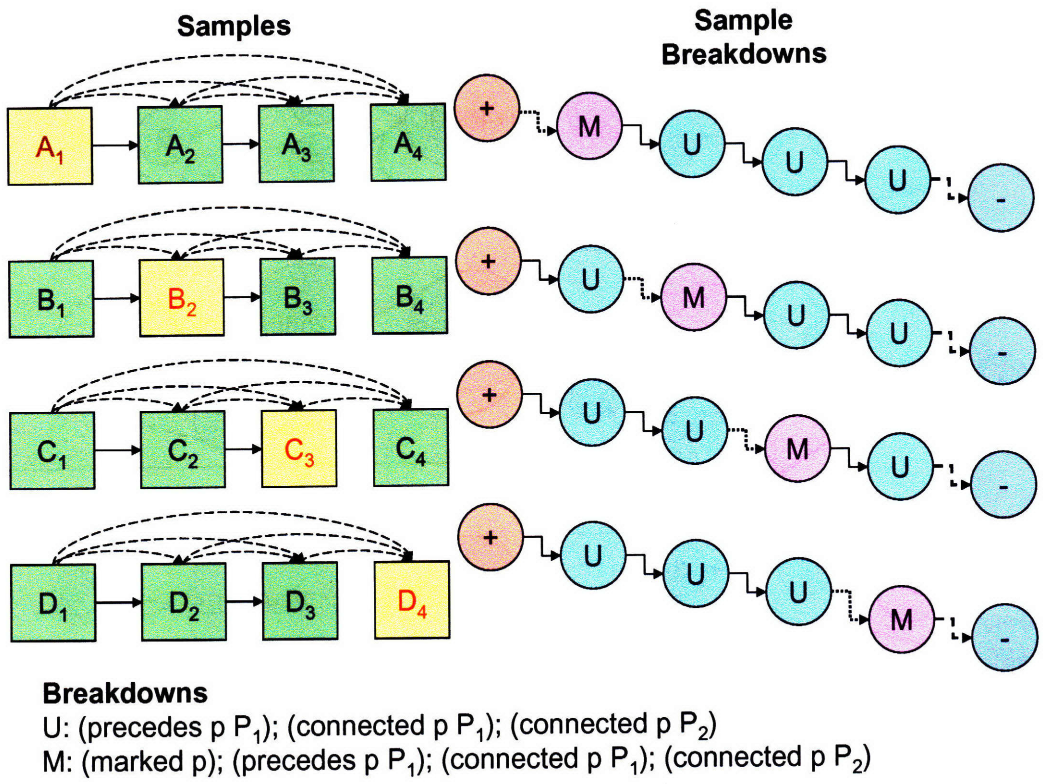

Figure 3-9: All marked lists of size 4 with their decompositions represented as state machines.

In the algorithm, the nodes of the finite-state machine are continually merged and unified by a set of unification rules until there are no more rules to apply. For example, consider the marked lists in Figure 3-9. These are all of the lists and sub-lists for the examples of size 4. There are two breakdown types, one where the pivot is the marked item and one where the pivot is not marked. These correspond to the dashed and the solid arcs respectively. We label them M and U. The empty sub-instance is marked with a double circle.

Unification of the nodes in the finite-state machine is the main operator in the algorithm. Two nodes A and B are unified by merging them into a common node AB. All arcs that led to either A or B now point to AB. Furthermore, any nodes pointed to by the same type of arc from A or B are recursively unified in turn. Figure 3-10 shows the unification process step by step. First, nodes A and B are unified.

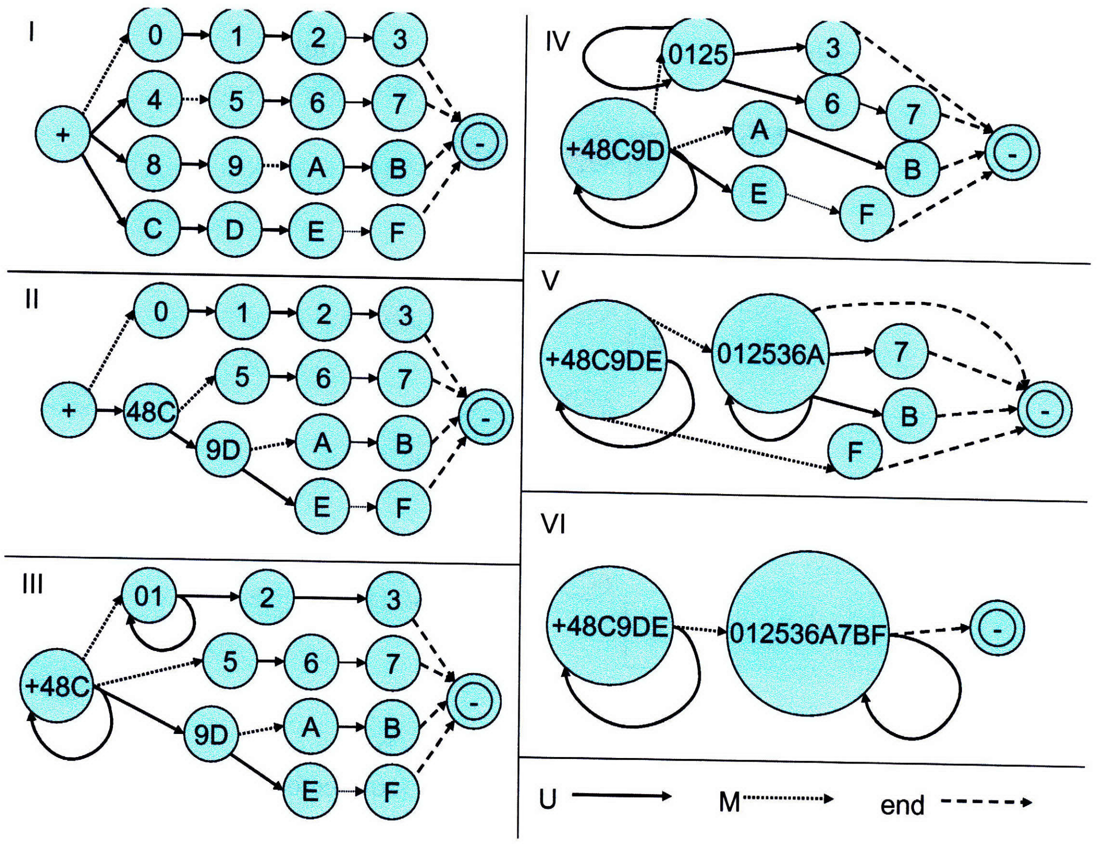

Figure 3-10: Sample breakdown unification step-by-step. The output machine has two states.

Then, because C and D are both pointed at by a dotted arc, they are unified in turn. Note that if there is an arc from A to B, when they are merged, the arc becomes a self-arc. If either of A or B are end nodes, the merged node AB is an end node.

At the beginning of grammar unification, the start nodes for each of the examples are all unified together. Panel I of Figure 3-10 shows the unified state machine for the marked lists of Figure 3-9. To decide which nodes to unify further, we select a look-ahead parameter k. A production string starting at node A is a sequence of arcs obtained by following a path in the finite state machine. The production string is the sequence of breakdown types for the arcs on the path. To handle termination, we create a supplementary node $-$. All end nodes have an arc to $-$ with the arc type *end*.

We denote $Prod(A, d)$ to be the set of production strings starting at A which

Algorithm 3 Unification

```
Unify(InMachine, k) {
  OutMachine := InMachine;
  while (true) {
    (A,B) = argmax ((Node1, Node2) in (OutMachine, OutMachine)
                                   with Node1 != Node2,
                    sim(Node1, Node2));
    if (sim(A,B) <= k) {
      break;
    }
    else {
      OutMachine := Unify(OutMachine, A, B);
    }
  }
  return OutMachine;
}
```

have length d or which terminate at $-$. For example, in Panel II of Figure 3-10, all production strings from node $48C$ of length 2 are $\{MU, UU, UM\}$. For every pair of nodes A and B, we calculate the maximum look-ahead similarity $sim(A, B)$. This is the maximum d such that $Prod(A, d) = Prod(B, d)$. It is the maximum d so that A and B are the same looking ahead d steps. $sim(A, B)$ can be infinite.

The unification algorithm, given look-ahead parameter k, is to find nodes A and B with the largest $sim(A, B)$ for which $sim(A, B) \geq k$ and merge them. The algorithm does this until there are no more nodes to merge.

The output is a small finite-state machine that is consistent with the decomposition discovered for the samples. With this finite-state machine, it is simple to construct a grammar corresponding to it. (A brief note: while the set of text string languages producable with grammars is strictly larger than the set of languages producable with state machines, for our logical worlds, grammars and state machines can produce the same world sets).

Consider our marked list example. When we run the unification algorithm with $k = 2$, it produces the state machine shown in Figure 4 with two states A and B. We associate these states with two non-terminals A_{in} and B_{in}. A_{in} is the top non-terminal. It can be broken down by breakdown M or U. Thus, we define $A_M \rightarrow$

(M, B_{in}) and $A_U \rightarrow (U, A_{in})$. When A breaks down by M, the next node is B, so A_M's sub-production is B_{in}. Similarly, when A breaks down by N, the sub-production for A_U is A_{in}again. A_{in} is produced as the disjunction of $A_M$and A_U. B is either ϵ or it breaks down via U into B again. Thus, B_{in} is a disjunction of B_U and ϵ. B_U, in turn, breaks down via U into B_{in}. All of this is summarized in the grammar below, including the breakdown information. Only the true relations of the breakdown are shown:

$$\begin{aligned}
A_{in} &::= A_M \mid A_U \\
A_M &::= piv; \{P_1, P_2\} \rightarrow B_{in}; \{(marked\ piv), (precedes\ piv\ P_1), \\
&\qquad (connected\ piv\ P_1), (connected\ piv\ P_2)\} \\
A_U &::= piv; \{P_1, P_2\} \rightarrow A_{in}; \{(precedes\ piv\ P_1), (connected\ piv\ P_1), \\
&\qquad (connected\ piv\ P_2)\} \\
B_{in} &::= B_U \mid \epsilon \\
B_U &::= piv; \{P_1, P_2\} \rightarrow B_{in}; \{(precedes\ piv\ P_1), (connected\ piv\ P_1), \\
&\qquad (connected\ piv\ P_2)\}
\end{aligned} \tag{3.5}$$

This is the same grammar as Formula 3.2. So, for this example, grammar induction by unification proved correct. Now, we can assign constraining quantified formulas to the non-terminals, and thus prove its suitability to represent the marked list quantified formula 4.1.

3.3.5 Assigning constraining formulas to non-terminals

Left-side formulas

Form2Grammar assigns formulas to both the left-hand non-terminals and the right-hand productions. Then, through a sequence of automatic proof steps, the system can show that any formula on the left-side can be produced by the corresponding formula on its right side. Producing the constraining quantified formulas for non-terminals in the grammar is a complex process whose workings are not necessary for a good understanding of the system. This section can be skimmed on a first reading.

Let us first consider how the left-hand formulas are made. To prove a grammar correct, one only needs to produce left-hand formulas for *in* non-terminals. These are the non-terminals that a decomposition production would reference to produce a component in its breakdown. The top non-terminal is also an *in* non-terminal. In our marked-list grammar, there are two *in* non-terminals, A_{in} and B_{in}.

For the top non-terminal, the constraining quantified formula is S, the given formula that Form2Grammar is trying to model. In the marked list case, the constraining formula for A_{in} is equation 4.1, which will be called ϕ_A. For a non-top *in* non-terminal NT_{in}, Form2Grammar augments the production in a specific way to connect it back to the top non-terminal. That is, it finds a fixed, concrete set of objects X so that X in combination with NT_{in} is the top non-terminal Top_{in}. In our example, B_{in} is a production rule for a list with no marked elements. It can be turned into a list with a marked element by adding one object at the front of the list. Formulaically, the concrete set X is made by finding the shortest derivation of the top non-terminal Top_{in} from a non-top non-terminal NT_{in}.

In our example, the derivation of an A_{in} from a B_{in} occurs in the following way:

$$\begin{aligned} A_{in} &\leftarrow A_M \\ A_M &\leftarrow \{(marked\ p), (precedes\ p\ P_1), (connected\ p\ P_1), (connected\ p\ P_2)\} \\ \{P_1, P_2\} &\leftarrow B_{in} \end{aligned}$$

So, the constraining quantified formula for B_{in} is then the formula $\phi_B$such that

$$\begin{aligned} &\phi_B : \{P_1, P_2\} \wedge (marked\ p) \wedge \forall y : P_1 (precedes\ p\ y) \wedge \quad \Rightarrow \quad \phi_A : \{p, P_1, P_2\} \\ &\forall y : P_1 (connected\ p\ y) \wedge \ldots \wedge \forall y : P_2 \neg (connected\ y\ p) \end{aligned}$$

That is, ϕ_B must be the formula over $\{P_1, P_2\}$ so that when ϕ_B is combined with the 9 breakdown formulas of M, the resulting formula satisfies ϕ_A over $\{p, P_1, P_2\}$.

To derive ϕ_B, the formula for the unmarked remainder list, we start with ϕ_A and break out the concrete object p. Breaking out points from formulas is discussed in Section 2.3. We also quantify ϕ_A into the disjoint set partition P_1 and P_2. Form2Grammar adds the breakdown formulas relating p to P_1 and P_2 and then simplifies. Finally, all of the formulas containing p are removed, replaced with T. The result is ϕ_B. This derivation will be examined in detail for the formula $\forall x$ *AllBut*(1) y $\neg$(*precedes* x y). This is the same as $\forall x U y$(*precedes* x y). Taking out p yields

$$\begin{aligned} & \quad \mathit{AllBut}(1)\ x : \{P_1, P_2\}\ \neg(\mathit{precedes}\ p\ y) \\ \wedge \quad \forall x : \{P_1, P_2\} & \quad (\neg(\mathit{precedes}\ x\ p)\ \wedge \\ & \qquad \mathit{AllBut}(1)\ y : \{P_1, P_2\}\ \neg(\mathit{precedes}\ x\ y)) \\ & \quad \vee\ ((\mathit{precedex}\ x\ p)\ \wedge\ \forall y : \{P_1, P_2\}\ \neg(\mathit{precedes}\ x\ y)) \end{aligned} \tag{3.6}$$

When the breakdown formulas

$\{(\mathit{precedes}\ p\ P_1), \neg(\mathit{precedes}\ p\ P_2), \neg(\mathit{precedes}\ P_1\ p), \neg(\mathit{precedes}\ P_2\ p)\}$

are added and simplified, this formula becomes:

$$\begin{aligned} & (\mathit{Unique}\ P_1) \\ \wedge\ & \forall x : \{P_1, P_2\}\ \mathit{AllBut}(1)\ y : \{P_1, P_2\}\ \neg(\mathit{precedes}\ x\ y) \end{aligned}$$

(*Unique* P_1) means that P_1has at most one element. Doing this derivation for the entire marked list formula ϕ_A, one obtains this formula for ϕ_B:

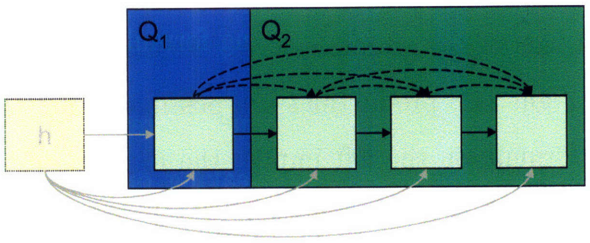

Figure 3-11: An example interface partition for an unmarked list. The constraining formula is defined in terms of Q_1, Q_2, and an external term h, which is eliminated in the final constraining formula.

$$\begin{aligned} & (Unique\ Q_1) \\ \wedge\ & \forall x : Q_2 \exists y : \{Q_1, Q_2\}\ (connected\ y\ x) \\ \wedge\ & \forall x : \{Q_1, Q_2\}\ \neg(marked\ x) \\ \wedge\ & listDef : \{Q_1, Q_2\} \end{aligned} \tag{3.7}$$

listDef is the list formula, formula 3.3, quantified over the set partition $\{Q_1, Q_2\}$, which has been renamed from $\{P_1, P_2\}$. Note that this formula is still defined with respect to $Q_1$and Q_2. Q_1 and Q_2 are called the *interface partition* to non-terminal B_{in}. A picture of the interface partition for a list with no marked elements is shown in Figure 1.

For non-*in* non-terminals (e.g., A_M, A_U, B_U), the left-side formula is copied from the right side formula. Thus, there is nothing to prove for non-*in* non-terminals.

Mapping Interface Partitions

Interface partitions add a level of subtle complexity to the grammar induction. Interface partitions occur any time there is a data structure which does not decompose into smaller versions of itself. For examples, lists and trees both decompose into smaller lists and trees. However, circularly linked lists and marked lists both decompose into structures that are not circularly linked lists or marked lists.

Let us consider how to prove a constraining grammar correct. When the *in* non-terminals on the left side of the grammar have formulas, we must produce formulas to indicate that the right side productions are possible. The theorem provers must then show that the formula on the left implies the formula on the right. The most complex non-terminal production is a decomposition. Recall that the general form for a decomposition is:

$$NT \ ::= piv; \{\{A_{1j}\} \rightarrow NT_1, \ldots, \{A_{mj}\} \rightarrow NT_m\}; (R\ piv\ A_{ij}),\ (R\ A_{ij}\ A_{kj})$$

Let ϕ_i be the left-side formula for non-terminal NT_i, let ϕ_{NT} be the left side non-terminal for the whole production, and let χ be the formula that is the conjunction of the cross relations between *piv* and A_{ij}, and between A_{ij} and A_{kj} in different sub-productions. The goal is to show that for any world satisfying ϕ_{NT}, there exists a partition $\{piv, A_{ij}\}$ such that $\phi_{RIGHT} : \{piv, A_{ij}\} = \phi_1 : \{A_{1j}\} \wedge \ldots \wedge \phi_m : \{A_{mj}\} \wedge \chi : \{piv, A_{ij}\}$ holds.

A technical issue is to map the interfaces of formulas ϕ_i with the interface for ϕ_{RIGHT}. ϕ_i may be defined with some interface partition $\{Q_i'\}$ and ϕ_{RIGHT} may be defined with respect to another interface partition $\{Q_i\}$ and there is no way to know, a priori, how these interfaces relate to each other. Without knowing this relationship, it is impossible to prove that ϕ_{RIGHT} implies ϕ_i.

In the marked list example, B_U is recursively defined in terms of B_{in}. The quantified formula $\phi_{B_U}$is ϕ_B for some partition $\{Q_1^L, Q_2^L\}$. Meanwhile, $B_{in}$is also defined by ϕ_B but over a different partition $\{Q_1^R, Q_2^R\}$. The problem is to relate these interfaces together, along with the breakdown partition $\{p, P_1, P_2\}$ which might be different from $\{Q_1^R, Q_2^R\}$.

To establish this relationship, Form2Grammar makes a guess based on the examples. Each sub-instance in the sample decomposition is broken up into its interface partition. This interface partition is compared with the breakdown partition in the parent node and the interface partition of the parent sub-instance. An example is

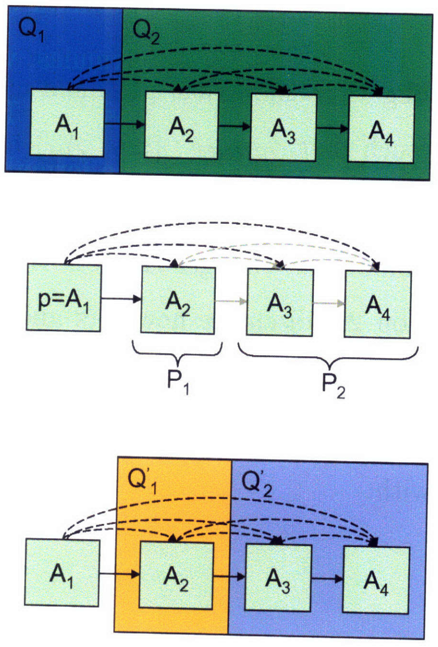

Figure 3-12: Three partitions of an unmarked list that is a sub-instance of a marked list. The interface partition $\{Q_1, Q_2\}$ is used to define the constraining formula. $\{p, P_1, P_2\}$is the breakdown partition in the sample decomposition. $\{Q'_1, Q'_2\}$ is the interface partition of the smaller instance $P_1 \cup P_2$. The samples are used to determine the relationships among these partitions. In this case, $Q'_1 = P_1 \cap Q_2$. $Q'_2 = P_2 \cap Q_2$. $Q_1 = \{p\}$. $Q_2 = (Q'_1 \cap P_1) \cup (Q'_2 \cap P_2)$.

shown in Figure 3-12.

The assumption is that each Q^L_iin the interface partition for a parent node ϕ_{NT} can be represented as the union of sets $\{Q^R_i \cap P_j\}$. Furthermore, each sub-interface Q^R_i is assumed to be representable by sets $\{Q^L_i \cap P_j\}$. For example, in the B_U decomposition, using examples, Form2Grammar discovers $Q^L_1 = \{p\}$ and $Q^L_2 = P_1 \cap Q^R_1 \bigcup P_2 \cap Q^R_2$. In the sub-production, $Q^R_1 = Q^L_2 \cap P_1$ and $Q^R_2 = Q^L_2 \cap P_2$. The assumption that interface partitions relate this way is strong, but it holds for all of the example cases that have been considered.

From now on, when considering right-side formulas ϕ_{RIGHT} and ϕ_i, it will be assumed that the interface partition sets of ϕ_i will have been replaced with intersections of breakdown partition sets A_{ij} and left-side interface partition sets Q^L_i.

Building Right-Side Formulas

The decomposition proof obligations show that a formula satisfying ϕ_{NT} breaks down into a partition $\{piv,\ A_{ij}\}$ such that $\phi_{RIGHT} : \{piv,\ A_{ij}\}$ holds.

To turn this partition existance statement into a quantified formula suitable for a theorem prover, Form2Grammar first changes the sets A_{ij}in ϕ_{RIGHT} into properties (relations of arity 1). Section 2.3 discusses how sets and properties are equivalent to each other. Then, the properties $(A_{ij}\ x)$ are replaced with the formula relating $A_{ij}$with the pivot point piv. For example, in the marked list example, $(P_1\ x)$ and $(P_2\ x)$ would be replaced with:

$$\begin{aligned}(P_1\ x) &= (\mathit{precedes}\ p\ x) \wedge (\mathit{connected}\ p\ x) \wedge \neg(\mathit{precedes}\ x\ p) \wedge \\ &\quad \neg(\mathit{connected}\ x\ p) \\ (P_2\ x) &= \neg(\mathit{precedes}\ p\ x) \wedge (\mathit{connected}\ p\ x) \wedge \neg(\mathit{precedes}\ x\ p) \wedge \\ &\quad \neg(\mathit{connected}\ x\ p)\end{aligned} \tag{3.8}$$

After the A_{ij} definitions have been substituted, the new formula ϕ'_{RIGHT} no longer has any references to sets A_{ij}.

Finally, the pivot point is quantified, so that $\exists piv\ \phi'_{RIGHT}$ is the formula that needs to be proven. Now, all of the sets and free variables of ϕ'_{RIGHT} relate to sets and free variables in the left side constraining formula ϕ_{NT}, and the theorem prover can be invoked.

The right-side formula for a disjunction is simple to state. If NT is produced as $NT_1 \mid \ldots \mid NT_m$, let $\phi_1, \ldots, \phi_m$ be the corresponding left-side formulas for $NT_1, \ldots, NT_m$. (If a non-terminal is not an *in* non-terminal, then its left-side formula is its right-side formula.) The right-side formula for NT, $\phi_{NT} = \phi_1 \vee \ldots \vee \phi_m$.

The right-side formula for the ϵ non-terminal is the size-possibilities formula $(None)$.

Let us compute right-side formulas for the marked list example. Refer to grammar

3.5. Form2Grammar will first form the right side for A_U. ϕ_A is the left-side formula for A_{in}. Then,

$$\phi_{A_U} \;=\; \exists p\; \phi'_{A_{in}} : \{P_1, P_2\} \wedge \chi'_U : \{p, P_1, P_2\},$$

where $\phi'_{A_{in}}$ is $\phi_{A_{in}}$ quantified over $\{P_1, P_2\}$ with P_1 and P_2 replaced with their definitions according to equation 3.8. χ'_U is defined similarly for the relations between p and $\{P_1, P_2\}$. However, with equation 3.8, most of the relations cancel each other and χ'_U simplifies to $\neg(marked\; p)$.

Similarly,

$$\begin{aligned}
\phi_{A_M} &= \exists p\; \phi'_B : \{P_1, P_2\} \wedge (marked\; p) \\
\phi_{B_U} &= \exists p\; \phi'_{B_{in}} : \{Q_1^R = Q_2^L \cap P_1, Q_2^R = Q_2^L \cap P_2\} \wedge \neg(marked\; p)
\end{aligned}$$

In this equation, the formula $\phi'_{B_{in}}$ must have its interface partition $\{Q_1^R, Q_2^R\}$ mapped to$\{Q_2^L \cap P_1, Q_2^L \cap P_2\}$, where $\{Q_1^L, Q_2^L\}$ is the interface partition for the left side $\phi_{B_{in}}$.

The right-side of A_{in},$\phi_{A_{in}}^{RIGHT} = \phi_{A_M} \vee \phi_{A_U}$. The right side of B_{in} is $\phi_{B_{in}}^{RIGHT} = \phi_{B_U} \vee (None)$. All of the constraints have now been defined. Now, Form2Grammar must prove that every left-side formula implies every right-side formula.

As a note, producing constraining formulas for grammars is a technically and logically complex process, whose working is not necessary for an understanding of Form2Grammar. The purpose of the constraining formulas is to produce a conjecture to prove. The conjecture is that the left side of a non-terminal breaks down according to the rule in the grammar.

3.3.6 Proofs: Simplification

Let us summarize where we are. Using examples, Form2Grammar has produced a simple grammar that fits all of them. The previous section shows how Form2Grammar assigns constraining quantified formulas to the non-terminals of the grammar and how interface partitions are related if they exist. The hypothesis, which Form2Grammar needs to prove, is that any world ω which satisfies the constraining formula, ϕ_{NT} on the left side of a non-terminal NT can be broken down according to the production rule on the right side. The decomposition production rule demands the existence of a suitable pivot as well as the satisfaction of constraining formulas on the partitioned subsets of ω, which will be called $\bar{\omega}$.

Overall, the proof of the correctness of the grammar is by structural induction. We assume by induction that $\bar{\omega}$ properly breaks down according to its non-terminal, $\overline{NT}$. So, by induction ω breaks down according to the production rule of NT, and this production rule accounts for every object and every relation in ω. Thus, if Form2Grammar can prove that the left-side formula implies the right-side formula, then it will have shown that the constraining grammar is fully correct. In the case of the marked list, there are two proof obligations to perform:

$$\begin{aligned}
\phi_A &\Rightarrow (\exists p\ \phi'_A : \{P_1, P_2\} \land \neg(\mathit{marked}\ p)) \lor \\
&\quad\ (\exists p\ \phi'_B : \{P_1, P_2\} \land (\mathit{marked}\ p)) \\
\phi_B &\Rightarrow (\exists p\ \phi'_B : \{P_1, P_2\} \land \neg(\mathit{marked}\ p)) \lor \\
&\quad\ (\mathit{None})
\end{aligned}$$

From a practical perspective, however, trying to prove these theorems in one step is intractable. The formulas ϕ'_A and ϕ'_B, when written out, are quite large. To prove the theorems in a reasonable amount of time, Form2Grammar first needs to do a simplification and then prove the existence of the pivot p.

Consider $\phi_A : \{P_1, P_2, p\}$, i.e. before P_1 and P_2 are converted to properties. A large part of this formula is the definition of a list over $\{P_1, P_2, p\}$. $\phi_A : \{P_1, P_2\}$ has the same list axioms over P_1 and P_2. There are thus many redundant requirements in $\phi_A : \{P_1, P_2\}$ that can be removed. So, to start the proof, Form2Grammar first simplifies $\phi_A : \{P_1, P_2\} \wedge \chi_U : \{p, P_1, P_2\}$ given $\phi_A : \{P_1, P_2, p\}$. Then, Form2Grammar takes out P_1 and P_2 and add $\exists p$. In practice, simplification often yields tremendous reductions in the complexity of formulas to be proved.

In general, Form2Grammar needs to show $\phi_{RIGHT} = \phi_{NT_1} : \{A_{1j}\} \wedge \ldots \wedge \phi_{NT_m} : \{A_{mj}\} \wedge \chi : \{piv, A_{ij}\}$ for some partition $\{piv, A_{ij}\}$. To make this easier, Form2Grammar simplifies ϕ_{RIGHT} given $\phi_{NT} : \{A_{ij}, piv\}$. That is, we take the given formula ϕ_{NT} and expand it over the partition $\{A_{ij}, piv\}$. The expansion-over-partition procedure is described in section 2.3.

If ϕ_{NT} has interface partition sets, these sets are replaced with the appropriate union of intersections from $A_{ij} \cap Q'_k$, the breakdown partition and sub-production interface partition sets.

In the marked list example, there are three breakdown decompositions, A_M, A_U, and B_U. For A_M, the task is to simplify $\phi_B : \{Q_1 = P_1, Q_2 = P_2\} \wedge \chi_M : \{p, P_1, P_2\}$ given $\phi_A : \{P_1, P_2, p\}$. For A_U, Form2Grammar needs to simplify $\phi_A : \{P_1, P_2\} \wedge \chi_U : \{p, P_1, P_2\}$ given $\phi_A : \{P_1, P_2, p\}$. For B_U, Form2Grammar needs to simplify $\phi_B : \{Q_1^R = P_1, Q_2^R = P_2\} \wedge \chi_U : \{p, P_1, P_2\}$ given $\phi_B : \{Q_1^L = \{p\}, Q_2^L = P_1 \cup P_2\}$.

The simplifications come out to:

$$
\begin{aligned}
simp(\phi_{A_M}) = &\quad \forall x : P_2 \neg(connected\ x\ p) \\
\wedge &\quad \forall x : P_1(precedes\ p\ x) \\
\wedge &\quad \forall x : P_2 \neg(precedes\ p\ x) \\
\wedge &\quad \forall x : P_2 \neg(marked\ x) \\
\wedge &\quad \forall x : P_1 \neg(marked\ x) \\
simp(\phi_{A_U}) = &\quad \forall x : P_2 \neg(connected\ x\ p) \\
\wedge &\quad \forall x : P_1(precedes\ p\ x) \\
\wedge &\quad \forall x : P_2 \neg(precedes\ p\ x) \\
\wedge &\quad \exists x : P_1(marked\ x)\ \vee\ \exists x : P_2(marked\ x) \\
simp(\phi_{B_U}) = &\quad \forall x : P_2 \neg(connected\ x\ p) \\
\wedge &\quad \forall x : P_1(precedes\ p\ x) \\
\wedge &\quad \forall x : P_2 \neg(precedes\ a\ x)
\end{aligned}
$$

Note the extraordinary degree of simplification that can be achieved in this example. Such reduction is typical for the problems that have been encountered. Much of the redundancy has been reasoned out of the formulas, and now the existence of the pivot p can be proven tractably.

In fact, however, there can be even further simplification. For the decomposition production, Form2Grammar expands out the breakdown sets $\{P_1, P_2\}$ in terms of their pivots and then simplifies the expansion given the left-hand formula. In the marked list example, $P_1$and P_2 expand out via equation 3.8 for all three productions A_M, A_U, and B_U. For example, $\forall x : P_1 \neg(marked\ x)$ would expand to:

$$
\begin{aligned}
\forall x\ \neg(P_1\ x) \vee \neg(marked\ x) \quad = \quad & \forall x \neg(precedes\ p\ x) \vee (precedes\ x\ p) \vee \\
& \neg(connected\ p\ x) \vee (connected\ x\ p) \vee \\
& \neg(marked\ x)
\end{aligned}
$$

To ensure correct, the expansion must also include the expansion of the partition clause:

$$\begin{aligned}
\forall x\ (P_1\ x) \vee (P_2\ x) = \quad & \forall x\ ((\mathit{precedes}\ p\ x) \wedge \neg(\mathit{precedes}\ x\ p) \wedge (\mathit{connected}\ p\ x) \wedge \\
& \quad \neg(\mathit{connected}\ x\ p)) \\
\vee \quad & \forall x\ (\neg(\mathit{precedes}\ p\ x) \wedge \neg(\mathit{precedes}\ x\ p) \wedge (\mathit{connected}\ p\ x) \wedge \\
& \quad \neg(\mathit{connected}\ x\ p))
\end{aligned}$$

Form2Grammar then simplifies $\mathit{expand}_{P_1,P_2}(\mathit{simp}(\phi_{A_M}))$ given $\phi_A : \{p, \neg p\}$,

$\mathit{expand}_{P_1,P_2}(\mathit{simp}(\phi_{A_U})$ given $\phi_A : \{p, \neg p\}$, and $\mathit{expand}_{P_1,P_2}(\mathit{simp}(\phi_{B_U})) : \{Q_2^L\}$ given $\phi_B : \{Q_1^L = \{p\}, Q_2^L\}$. $\phi_A : \{p, \neg p\}$ means ϕ_A with the pivot point broken out as in formula 3.6. For B_U, the interfaces have been matched to that the pivot point is assumed to be in Q_1^L of the left-side interface partition.

The second simplification yields the following formulas for A_M, A_U, and B_U.

$$\begin{aligned}
\mathit{simp}^2(\phi_{A_M}) = \quad & \forall x\ \neg(\mathit{marked}\ x) \\
\wedge \quad & \forall x\ \neg(\mathit{connected}\ x\ p) \\
\mathit{simp}^2(\phi_{A_U}) = \quad & \exists x\ (\mathit{marked}\ x) \\
\wedge \quad & \forall x\ \neg(\mathit{connected}\ x\ p) \\
\mathit{simp}^2(\phi_{B_U}) = \quad & \forall x : Q_2^L\ \neg(\mathit{connected}\ x\ p)
\end{aligned}$$

Now, the formulas are in a form where Form2Grammar can attempt to prove the existence of p given the left-side formula. In the marked list example, there are now two theorems to prove:

$$\begin{aligned}
\phi_A \Rightarrow &\quad (\exists p\, \forall x\, \neg(\mathit{marked}\ x)\ \wedge\ \forall x\, \neg(\mathit{connected}\ x\ p)) \\
&\vee\ (\exists p\, \exists x(\mathit{marked}\ x)\ \wedge\ \forall x\, \neg(\mathit{connected}\ x\ p)) \\
\phi_B : \{Q_1^L, Q_2^L\} \Rightarrow &\quad (\exists p : Q_1^L\ \forall x : Q_2^L\ \neg(\mathit{connected}\ x\ p)) \\
&\vee\ ((\mathit{None}\ Q_1^L)\ \wedge\ (\mathit{None}\ Q_2^L))
\end{aligned}$$

Finally, the simplification algorithm is run one last time to extract the last bits of redundancy. This yields the following obligations:

$$\begin{aligned}
\phi_A &\Rightarrow \exists p \forall x \neg(\mathit{connected}\ x\ p) \\
\phi_B : \{Q_1^L, Q_2^L\} &\Rightarrow \exists p : Q_1^L\ \forall x : Q_2^L \neg(\mathit{connected}\ x\ p)\ \vee\ (\mathit{None}\ Q_2^L) \qquad (3.9)
\end{aligned}$$

The first obligation states that there must exist an element in the marked list that is the first element, one that nothing else is connected to. The second obligation states that an unmarked sub-list is either null, or it has one element in the Q_1^L interface partition set that nothing else is connected to. Both of these obligations are facts that can only be proven by mathematical induction. That is, pure first-order logic techniques like resolution are not sufficient. Section 3.3.8 will present a general purpose theorem prover that is capable of proving the above two theorems without any special rules or induction axioms.

3.3.7 Simplification Theorem Proving

As can be seen in the above marked list example, the simplification routines are extremely important and powerful, incorporating some deep reasoning and chaining. This section describes the simplification algorithms used to make the final theorem proving more tractable. They include a shallow subsumption algorithm, a shallow first-pass simplifier and a deep simplifier that can do complex inferences including rule chaining.

One critical restriction is that, because of the quantified formula representation, Form2Grammar cannot use resolution and unification. Because of the implicit not-equal assumptions in formulas such $\forall x \neg(\textit{connected } x \ p)$, one cannot simply replace the quantified variable x with a free variable $?x$ that could then be unified to anything else. Unification is not necessarily impossible, but the bookkeeping becomes exceedingly complex, and the benefits of resolution are outweighed by the costs. The deep simplifier thus does simplification by tree pruning, which is a technique that can perform many of the inferences that the resolution rule can do.

Subsumption

The central mechanism in all of the simplifiers is a fast subsumption test. Given two formulas p and q, the subsumption function tests whether $p \Rightarrow q$. If the function returns $True$, then $p \Rightarrow q$. However, if it returns $False$, there is no guarantee that $p \not\Rightarrow q$. Let $\mapsto$denote the subsumption operator. In the case statement in Figure 3-13, the first rule that matches p and q on the left side is the one that is applied. & and || are the boolean AND and OR operators respectively. The last rule is a catch-all for formula pairs that do not match.

In quantified formulas $p(x)$ and $q(y)$, the substitutions $p[x|z]$ and $q[y|z]$ refers to the new term z, which is a fresh term not contained anywhere in p or q. Subsumption can be applied to quantified formulas quantified over sets. When this happens, the subsumption can include a size possibilities context derived from neighboring formulas (e.g. whether a certain set A has at least one element, which is derived from a formula like $\exists x : A\ (marked\ x)$).

To improve performance, the subsumption function includes a cache of the most recently solved subsumptions. This makes the function fast and predictable.

Shallow simplification

Shallow simplification is denoted by p/q, which means the simplification of p given q. p/q travels through the tree of p and eliminates branches that are subsumed in some way by q. If $q \mapsto p$, then $p/q = T$. If $q \mapsto \neg p$, then $p/q = F$. The complex cases are

$$
\begin{aligned}
F \mapsto q &= True \\
p \mapsto T &= True \\
p \mapsto p &= True \\
p \mapsto (q_1 \wedge \ldots \wedge q_m) &= (p \mapsto q_1)\&\ldots\&(p \mapsto q_m) \\
(p_1 \vee \ldots \vee p_k) \mapsto q &= (p_1 \mapsto q)\&\ldots\&(p_k \mapsto q) \\
(p_1 \wedge \ldots \wedge p_k) \mapsto (q_1 \vee \ldots \vee q_m) &= (p_1 \wedge \ldots \wedge p_k \mapsto q_1)||\ldots||(p_1 \wedge \ldots \wedge p_k \mapsto q_m)|| \\
& (p_1 \mapsto q_1 \vee \ldots \vee q_m)||\ldots||(p_k \mapsto q_1 \vee \ldots \vee q_m) \\
(p_1 \wedge \ldots \wedge p_k) \mapsto q &= (p_1 \mapsto q)||\ldots||(p_k \mapsto q) \\
p \mapsto (q_1 \vee \ldots \vee q_m) &= (p \mapsto q_1)||\ldots||(p \mapsto q_m) \\
AtLeast(n)\ x : A\ p(x) \mapsto &= n \geq m\ \&\ A \subseteq B\ \&\ p[x|z] \mapsto q[y|z] \\
AtLeast(m)\ y : B\ q(y) & \\
AtLeast(AllBut(n))\ x : A\ p(x) \mapsto &= n \geq m\ \&\ A \supseteq B\ \&\ p[x|z] \mapsto q[y|z] \\
AtLeast(AllBut(m))\ y : B\ q(y) & \\
AtLeast(AllBut(n))\ x : A\ p(x) \mapsto &= |A| \geq n + m\ \&\ A \subseteq B\ \&\ p[x|z] \mapsto q[y|z] \\
AtLeast(m)\ y : B\ q(y) & \\
AtLeast(n)\ x : A\ p(x) \mapsto &= |A| \leq n + m\ \&\ A \supseteq B\ \&\ p[x|z] \mapsto q[y|z] \\
AtLeast(AllBut(m))\ y : B\ q(y) & \\
p \mapsto q &= False
\end{aligned}
$$

Figure 3-13: Ssubsumption rules

when p is an and, an or, or a quantification.

First, let $p = p_1 \wedge \ldots \wedge p_k$. The procedure is as follows:

```
$s_i := p_i$
**for** $i = 1 \ldots k$ {
    $s_i := p_i/(q \wedge \bigwedge_{j \neq i} s_j)$
}
**return** $s_1 \wedge \ldots \wedge s_k$
```

For each conjunct p_i, the given formula q is first simplified relative to p_i's neighboring conjuncts. Then, p_i is simplified relative to the new q'. This complication is added to allow for a first level of inference. For example $(R \wedge S)/(\neg R \vee S) \; = R$.

Similarly, let $p = p_1 \vee \ldots \vee p_k$. Then, the procedure is:

```
$s_i := p_i$
**for** $i = 1 \ldots k$ {
    $s_i \; := p_i/q \wedge \bigwedge_{j \neq i} \neg s_j$
}
**return** $s_1 \vee \ldots \vee s_k$
```

For a quantification, $Qx : A \; p(x)/q = Qz : A \; (p[x|z]/\mathit{extract}(q, A, z))$. *extract* is a utility function that finds and conjoins all of the $\forall y : B \; r(y)$ sub-formulas in q where $B \supseteq A$. It removes the $\forall$ and replaces y with z. *extract* can pull out sub-formulas that are under other quantifiers. For example, *extract*($\forall x : A \forall y : A \neg$(*precedes* x y) $\vee$ (*connected* x y)) $=$

$$\begin{aligned} & \forall y : A\neg(\mathit{precedes}\ z\ y) \vee (\mathit{connected}\ z\ y) \\ \wedge\ & \forall x : A\neg(\mathit{precedes}\ x\ z) \vee (\mathit{connected}\ x\ z) \end{aligned}$$

Like subsumption, shallow simplification also caches recently solved reductions, so that the function can take advantage of memoization. Shallow simplification usually finishes in a short amount of time.

Deep Simplification

Once shallow simplification has taken the most obvious redundancies out of a formula, deep simplification can run. This usually takes all redundancies out that can be deduced using standard logical inference. However, because it can chain indefinitely, it can sometimes take a considerable amount of time to finish. We denote the deep simplification operation as $p \oslash q$. Deep simplification actually works by simplifying both p and q at the same time. An intermediate product, called the *attachment,* $q \oplus p$, is the joint simplification of $q \wedge p$. If $q \oplus p = q' \wedge p'$, then $p \oslash q = p'$

When a single formula p needs to be deep simplified, this is $p \oslash T$.

These are the rules:

If $q \mapsto p$, $q \oplus p = q$. If $q \mapsto \neg p$, $q \oplus p = F$.

For conjunctions, $q \oplus (p_1 \wedge \ldots \wedge p_k) = (q \oplus p_1) \oplus (p_2 \wedge \ldots \wedge p_k)$.

For every other formula type, disjunctions, quantifications, and ground relations, the attachment may proceed through several rounds. Each round produces a hypothesis formula h, which might be simplified further.

Consider $p = p_1 \vee \ldots \vee p_k$. Form2Grammar initializes $s_i = p_i$. Then, it sets $s_k = p_k \oslash (q \oplus \neg s_1 \oplus \ldots \oplus \neg s_{k-1})$.

Similarly, it sets $s_{k-1} = p_{k-1} \oslash (q \oplus \neg s_1 \oplus \ldots \oplus \neg s_{k-2} \oplus \neg s_k)$. In a loop, it then sets s_i down to $s_1 = p_1 \oslash (q \oplus \neg s_2 \oplus \ldots \oplus \neg s_k)$. The candidate output is then $s_1 \vee \ldots \vee s_k$. The fact used to simplify the disjunction is that whenever one wants to simplify p_k, one can assume that $\neg p_1 \wedge \ldots \wedge \neg p_{k-1}$, because any truth outside of this negation is

```
$hypothesizeSimplification(p_1 \vee \ldots \vee p_k, q)$ {
  $s_i = p_i$
  $done =$ **false**
  **while** $(\neg done)$ {
    $s'_i := s_i$
    $done =$ **true**
    **for** $i = 1 \ldots k$ {
      $s'_i := s'_i \oslash (q \oplus \neg s'_1 \oplus \ldots \oplus s'_{i-1} \oplus s'_{i+1} \oplus \ldots \oplus s'_k)$
      **if** $s'_i \not\mapsto s_i$
        $done =$ **false**
    }
    $s_i := s'_i$
  }
  $h := s_1 \vee \ldots \vee s_k$
}
```

Figure 3-14: The first pass of the deep simplification algorithm

handled by the other formulas in the disjunction $p_1 \vee \ldots \vee p_{k-1}$. The s_i are used to make sure that truth overlaps are only applied one way. p_2 can remove the common part $p_1 \wedge p_2$, but p_1 must retain this overlapping piece.

For an example, consider the simplification of $p \vee p$. The second p assumes $\neg p$. Thus, $s_2 = F$. Thus, the first p is simplified given $\neg F = T$. Thus, the output is $p \vee F = p$.

If $s_i \not\mapsto p_i$, then in some way, s_i has become weaker. For example, if p_i was originally $X \wedge Y$ and s_i became X, then s_i is a weaker condition. At the end of the k simplifications, if one of the s_i has become weaker, the loop is run again. Potentially, information has been gained that could further simplify some s_j coming after s_i. The loop is run until every $s_i^{post} \mapsto s_i^{pre}$. The method is summarized in Figure 3-14.

If p is a quantification $Qx : A\ p(x)$, then $h = Qz : A\ (p[x|z] \oslash extract(q, A, z))$. If p is a primitive relation, then $h = p$.

For all formula types, once the hypothesis h has been proposed, the given formula q is *shallow* simplified relative to h, producing $q' = q/h$. This simplification has the

```
$p \oslash q(subsumeOkay)$ {
  $h = hypothesizeSimplification(p, q, subsumeOkay)$
  $q' = q/h$
  **if** $q \not\mapsto q'$ {
    $r = q' \oslash T(\textbf{true})$
    **return** $h \oslash r(\textbf{false})$
  }
  **else** {
    **return** $h$
  }
}
```

Figure 3-15: The full deep simplification algorithm

potential to strengthen q, i.e. $q \not\mapsto q'$. If $q \mapsto q'$, then deep simplification is done: $q' \wedge h = q \oplus p$ and $h = p \oslash q$. Otherwise, if q' is stronger than q, h potentially can be simplified further.

First, q'must be re-simplified by itself, $r = q' \oslash T$. Then, h is re-simplified, producing $r \oplus h$ and $h \oslash r$ as the attachment and the deep simplification respectively. The whole algorithm is summarized in Figure 3-15.

The indefinite rebuilding and resimplification allows the algorithm to follow an arbitrary-length chain of rules. It also means that the function can take an unpredictably long time to terminate. Thus, deep simplification is only practical when the formula p is small.

Certain cases require special care. For example, consider $p = (Exists)$ and $q = (None) \vee \exists x\ (start\ x)$. Initially, p cannot be simplified, so the first hypothesis $h = (Exists)$. Then, the simplification of q given h, $q' = q/h = \exists x\ (start\ x)$. But, then $(Exists) \oslash \exists x\ (start\ x) = T$. This is an incorrect simplification, because $q \not\Rightarrow p$. The resulting $q \oplus p = \exists x\ (start\ x)$ is correct, but $p \oslash q$, calculated in the naive way, is wrong. Thus, deep simplification has the additional context parameter *subsumeOkay*. If the hypothesis h strengthens the given q to q', then it is not allowed for q' to replace any branches of h to T. It can only set branches to F. That is, q' is only allowed to strengthen h. It cannot weaken it in any way.

To make deep simplification tractable, the algorithm contains many engineering heuristics and short-cuts. Mainly, it tries to re-use computation as much as possible by caching solutions in a canonical form. Also, when producing r, the rebuilt q', deep simplification re-simplifies only those branches of q' that have been strengthened.

3.3.8 Theorem Proving by Abstract Induction

The previous two sections have shown how Form2Grammar can perform standard first-order reasoning over quantified formulas without the need for unification and reasoning about equality. The marked list example has shown that deep simplification can reduce a very complicated theorem obligation into a more manageable one. One of these obligations is, given the formula for a marked list, prove $\exists p \forall x \neg(\mathit{connected}\ x\ p)$. This is a formula that cannot be proven using ordinary first-order reasoning. It is a fact that is only true for finite models of the marked list formula, and thus it must be proven by some type of mathematical induction.

In this section, we will introduce the final theorem prover, which is capable of discharging this last proof obligation. This theorem prover uses a novel technique which is called abstract induction. It is capable of proving theorems like $\phi_A \Rightarrow \exists p \forall x \neg(\mathit{connected}\ x\ p)$ with no additional information. That is, it does not need any special induction axioms, and it does not rely on any knowledge base. This makes it ideal for automatic structure learning and other parts of automatic program synthesis.

Proving that a theorem $p \Rightarrow q$ is equivalent to proving that $p \wedge \neg q$ is impossible. That is, there is no world ω such that $(p \wedge \neg q)(\omega) = T$. To disprove $p \wedge \neg q$, abstract induction exhaustively searches all possible worlds by forming world abstractions and evaluating them. The logic of abstract induction is complex, so it is important to keep in mind the intuition. To falsify a formula, small worlds are created, and for each world, the theorem prover shows that the world ω cannot be a subset of a world satisfying $p \wedge \neg q$. That is, the world ω cannot be extended to a larger world satisfying $p \wedge \neg q$. Worlds are shown to be impossible to extend when the process of extending them to satisfy $p \wedge \neg q$ results in an infinite loop. The infinite loop is discovered by making abstractions of worlds and showing that the abstractions repeat forever.

This intuition is demonstrated in Section 3.3.8 with the example of falsifying $\phi_A \wedge \forall p \exists x$ (*connected x p*). To understand how abstract induction works, this is the only thing to understand. The intricate logical mechanics below just make this intuition precise.

Definition A *world abstraction,* α is a triple (O, A, I) consisting of a set of concrete terms O, a set of wildcards A, and an interpretation I.

Abstractions correspond to sets of a world $\omega$of size at least 2. The interpretation assigns truth values to relations among concrete terms and abstractions. However, instead of taking values *True* and *False*, the interpretation takes values *All*, *None*, or *Exists* for any given relation $R(a,\ o)$.

A world abstraction $\alpha = (O, A, I)$ is a valid abstraction for world (O', I') iff the following hold: $O \subset O'$. There exists a surjective(onto) mapping $\theta : O' - O \rightarrow A$ such that $|\theta^{-1}(a)| \geq 2$. That is, every $o \in O'$that is not in O maps to some wildcard $a \in A$. Every wildcard $a \in A$ corresponds to a set of objects in $O' - O$ of size at least 2. Finally, let $R(a_1, \ldots, a_n)$ be a relation of α.

$$\begin{aligned}
I(R(a_1, \ldots, a_n)) = All &\Leftrightarrow \forall x_1 \rightarrow a_1 \forall x_2 \rightarrow a_2 \ldots \\
&\quad\ \forall x_n \rightarrow a_n I'(R(x_1, x_2, \ldots, x_n)) = T \\
I(R(a_1, \ldots, a_n)) = None &\Leftrightarrow \forall x_1 \rightarrow a_1 \ldots \forall x_n \rightarrow a_n I'(R(x_1, \ldots, x_n)) = F \\
I(R(a_1, \ldots, a_n)) = Exists &\Leftrightarrow \exists x_1 \rightarrow a_1 \ldots x_n \rightarrow a_n I'(R(x_1, x_2, \ldots, x_n)) = T \ \& \\
&\quad\ \exists x_1 \rightarrow a_1 \ldots \exists x_n \rightarrow a_n I'(R(x_1, \ldots, x_n)) = F
\end{aligned}$$

Relations in the interpretation I can be between concrete terms and wildcards, wildcards and wildcards, or entirely among concrete terms. For any given world $\omega = (O', I')$ and a term-wildcard pair (O, A) with $O' \rightarrow (O, A)$, there exists a unique interpretation I to make $\alpha = (O, A, I)$ an abstraction for ω. To clarify notation, $x \rightarrow a$ if a is a wildcard and $\theta(x) = a$ or if a is a concrete term and $a = x$. For example,

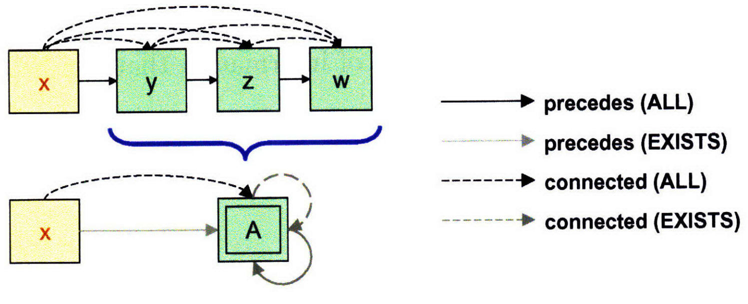

Figure 3-16: A sample of size 4 x, y, z, w and its abstraction. y, z, w is abstracted into the wildcard A.

consider the concrete world in Figure 3-16. It is a marked list with the first element marked. An abstraction for the list is shown below it. The abstraction consists of one concrete term x which is the marked first element. The wildcard consists of the rest of the list $a = \{y, z, w\}$. The interpretation I is defined as follows: $I((precedes\ x\ a)) = Exists$, $I((precedes\ a\ x)) = False$, $I((connected\ x\ a)) = True$, $I((connected\ a\ x)) = False$, $I((precedes\ a\ a)) = Exists$, $I((connected\ a\ a)) = Exists$. Note that wildcards have well-defined self-relations, since two objects mapping to the same wildcard will be related in a well-defined way.

Two abstractions $\alpha_1 = (O_1, A_1, I_1)$ and $\alpha_2 = (O_2, A_2, I_2)$ are considered to be equivalent iff there is a bijection $f : O_1 + A_1 \rightarrow O_2 + A_2$ such that $I_1(R(a_1, \ldots, a_n)) = I_2(R(f(a_1), f(a_2), \ldots, f(a_n))$ for all relations $R(a_1, \ldots, a_n)$. $\alpha_1$is considered to be a subset of α_2, written $\alpha_1 \leq \alpha_2$ iff there is an *injection*, a one-to-one map $f : O_1 + A_1 \rightarrow O_2 + A_2$ where $I_1(R(a_1, \ldots, a_n)) = I_2(R(f(a_1), \ldots, f(a_n))$. α_2 is called an *extension* of α_1.

For any abstraction α, we denote $\Omega(\alpha)$ to be the set of worlds ω for which α is an abstraction.

The semantics of ϕ distinguish between *All* and *Exists*. In the former case, $\phi(\alpha) = All$ if $\phi[X \rightarrow A](\omega) = T$ for all valid substitutions X for the free wildcard variables A. $\phi(\alpha) = Exists$ if for all worlds $\omega \in \Omega(\alpha)$, there exists a valid substitution $X \rightarrow A$ such that $\phi[X \rightarrow A](\omega) = T$. A substitution X is valid if it ensures that there are no duplicate terms created. That is X does not intersect the free terms of ϕ

and every distinct pair of free wildcard variables a_1 and a_2 is mapped to by distinct terms $x_1$and x_2 of ω. There are four types of information that can be gathered from a quantified formula $\phi(\alpha)$.

$$\begin{aligned}
All(\phi(\alpha)) &\Leftrightarrow \forall\omega \in \Omega(\alpha)\ \forall X \rightarrow Frees(\phi)\ \phi[X](\omega) = T \\
None(\phi(\alpha)) &\Leftrightarrow \forall\omega \in \Omega(\alpha)\ \forall X \rightarrow Frees(\phi)\ \phi[X](\omega) = F \\
ExistsT(\phi(\alpha)) &\Leftrightarrow \forall\omega \in \Omega(\alpha)\ \exists X \rightarrow Frees(\phi)\ \phi[X](\omega) = T \\
ExistsF(\phi(\alpha)) &\Leftrightarrow \forall\omega \in \Omega(\alpha)\ \exists X \rightarrow Frees(\phi)\ \phi[X](\omega) = F
\end{aligned}$$

From these four bits of information, there are six possible semantic values that $\phi(\alpha)$ can take on. The first case on the left side that matches $\phi(\alpha)$ determines its semantic value.

$$\begin{aligned}
All(\phi(\alpha)) &\Rightarrow \phi(\alpha) = All \\
ExistsT(\phi(\alpha)) \wedge ExistsF(\phi(\alpha)) &\Rightarrow \phi(\alpha) = ExistsTF \\
ExistsT(\phi(\alpha)) &\Rightarrow \phi(\alpha) = ExistsT \\
ExistsF(\phi(\alpha)) &\Rightarrow \phi(\alpha) = ExistsF \\
None(\phi(\alpha)) &\Rightarrow \phi(\alpha) = None \\
T &\Rightarrow \phi(\alpha) = Unknown
\end{aligned}$$

The top-level formula ϕ, which the prover tries to prove, has no free wildcard variables, so it can only evaluate to $All, None$, or $Unknown$.

Definition A *covering set* for a quantified formula ϕ is a set of abstractions $\mathcal{A} = \{\alpha_1, \ldots, \alpha_n\}$ such that every possible world $\omega \in \Omega(\phi)$ has an abstraction α for which $\alpha_i \leq \alpha$ and $\alpha_i \in \mathcal{A}$.

Definition A *dead-end* for a quantified formula ϕ is an abstraction α such that every extension $\alpha_{ex} \geq \alpha$ is false. That is, $\phi(\alpha_{ex}) = None$ for $\alpha_{ex} \geq \alpha$.

The abstract induction theorem prover shows that a quantified formula ϕ is impossible by producing a covering set $\mathcal{A} = \{\alpha_1, \ldots, \alpha_n\}$ in which every α_i is a dead-end. In this way, every world $\omega \in \Omega(\phi)$ will have abstractions α and α_i where $\omega \in \Omega(\alpha)$, $\alpha_i \in \mathcal{A}$, and $\alpha_i \leq \alpha$. Thus $\phi(\alpha) = F$ and thus $\phi(\omega) = F$. So, if the abstract induction theorem prover can find a dead-end covering set $\mathcal{A}$ for quantified formula ϕ, then it has proven that ϕ is impossible.

With the semantics properly grounded, it will now be shown how the theorem prover shows that a particular abstraction α is a dead-end. For a concrete example, consider proving $\exists z \forall x\ \neg(\mathit{connected}\ x\ z)$ given the list axioms. The negation of this formula is $\forall z \exists x\ (\mathit{connected}\ x\ z)$, which is added to the hypothesis formula (equation 4.1).

Calculating $\phi(\alpha)$

First, it will be shown how to compute $\phi(\alpha)$. Abstract induction tries to keep as much information as we can, so the logic is quite complicated. Howver, the mechanism for computing $\phi(\alpha)$ is not important for the rest of the theorem prover. This section should be skipped on a first reading. Let $\alpha = (O, A, I)$. Since $\phi(\alpha)$ is a logical quantity, operationally, abstract induction computes $\phi^*(\alpha)$, which is a conservative approximation to $\phi(\alpha)$ that can quickly be computed from the information at hand. The six semantic values $\{\mathit{All}, \mathit{ExistsTF}, \mathit{ExistsT}, \mathit{ExistsF}, \mathit{None}, \mathit{Unknown}\}$ are abbreviated to $\{A, \mathit{ETF}, \mathit{ET}, \mathit{EF}, N, U\}$.

A conservative definition for boolean operations on $\{T, F, \mathit{ETF}, \mathit{ET}, \mathit{EF}, U\}$ is the following. The first pattern that matches is the rule that is applied.

$$\begin{aligned}
N \wedge x &= N \\
A \wedge x &= x \\
EF \wedge x &= EF \\
ETF \wedge x &= EF \\
y \wedge x &= U \\
A \vee x &= A \\
N \vee x &= x \\
ET \vee x &= ET \\
ETF \vee x &= ET \\
\neg T &= F \\
\neg F &= T \\
\neg ETF &= ETF \\
\neg EF &= ET \\
\neg ET &= EF \\
\neg U &= U
\end{aligned}$$

When ϕ is a leaf relation $R(a_1, \ldots, a_n)$, then the evaluation is straightforward.

$$\begin{aligned}
I(R(a_1, \ldots, a_n)) = \mathit{All} &\Leftrightarrow \phi^*(\alpha) = A \\
I(R(a_1, \ldots, a_n)) = \mathit{None} &\Leftrightarrow \phi^*(\alpha) = N \\
I(R(a_1, \ldots, a_n)) = \mathit{Exists} &\Leftrightarrow \phi^*(\alpha) = ETF
\end{aligned}$$

When $\phi$is a conjunction or disjunction, $(\phi_1 \wedge \ldots \wedge \phi_n$ or $\phi_1 \vee \ldots \vee \phi_n)$, then $\phi^* = \phi_1^* \wedge \ldots \wedge \phi_n^*$ or $\phi^* = \phi_1^* \vee \ldots \vee \phi_n^*$. Evaluating quantifications is complicated. While adding up the effects of each possible substitution, the prover must keep track of eight integer quantities, one for each element of the set $Count = (\{min, max\}, \{minSub, maxSub\}, \{T, F\})$. The max values can be ∞. Assume $\phi = AtLeast(n)\ x\ \phi(x)$. Then,

$$\begin{aligned}
(min, minSub, T) \geq n &\Rightarrow \phi^*(\alpha) = A \\
(max, maxSub, T) < n &\Rightarrow \phi^*(\alpha) = N \\
(min, maxSub, T) \geq n \ \wedge\ (max, minSub, T) < n &\Rightarrow \phi^*(\alpha) = ETF \\
(min, maxSub, T) \geq n &\Rightarrow \phi^*(\alpha) = ET \\
(max, minSub, T) < n &\Rightarrow \phi^*(\alpha) = EF \\
T &\Rightarrow \phi^*(\alpha) = U
\end{aligned}$$

In the other case, assume $\phi = AtLeast(AllBut(n))\ x\ \phi(x)$. Then,

$$\begin{aligned}
(max, maxSub, F) \leq n &\Rightarrow \phi^*(\alpha) = A \\
(min, minSub, F) > n &\Rightarrow \phi^*(\alpha) = N \\
(min, maxSub, F) > n \wedge (max, minSub) \leq n &\Rightarrow \phi^*(\alpha) = ETF \\
(max, minSub, F) \leq n &\Rightarrow \phi^*(\alpha) = ET \\
(min, maxSub, F) > n &\Rightarrow \phi^*(\alpha) = EF \\
T &\Rightarrow \phi^*(\alpha) = U
\end{aligned}$$

The theorem prover keeps track of these eight quantities for each substitution $z \to x$ of $z \in O - Frees(\phi)$ and for each wildcard $a \in A$. For the wildcard, a fresh wildcard variable $a_0$is created and substituted for x to distinguish it from the other variables for a in ϕ. Here are the equations to update $Count$ when the concrete term z is substituted. There are 6 possible values $\phi^*[x|z](\alpha)$ can take on.

$$\begin{aligned}
\phi^*[x|z](\alpha) = A &\Rightarrow (x, y, T) \mathbin{+\!+} \\
N &\Rightarrow (x, y, F) \mathbin{+\!+} \\
ETF &\Rightarrow (max, y, z) \mathbin{+\!+} \\
&\quad\ (min, maxSub, T) = (min, minSub, T) + 1; \\
&\quad\ (min, maxSub, F) = (min, minSub, F) + 1 \\
ET &\Rightarrow (max, y, z) \mathbin{+\!+}; (min, maxSub, T) = (min, minSub, T) + 1 \\
EF &\Rightarrow (max, y, z) \mathbin{+\!+}; (min, maxSub, F) = (min, minSub, F) + 1 \\
U &\Rightarrow (max, y, z) \mathbin{+\!+}
\end{aligned}$$

The other possible substitution is with a wildcard variable a_0. One must be careful about counting if this is the third substitution of a variable for the wild card a. In that case, none of the (min, x, y) counts can be incremented. However, the usual case is:

$$\begin{aligned}
\phi^*[x|a](\alpha) = A &\Rightarrow (max, y, T) = \infty; (min, y, T) \mathbin{+\!+} \\
N &\Rightarrow (max, y, F) = \infty; (min, y, F) \mathbin{+\!+} \\
ETF &\Rightarrow (max, y, z) = \infty; (min, maxSub, T) = (min, minSub, T) + 1; \\
&\quad\ (min, maxSub, F) = (min, minSub, F) + 1 \\
ET &\Rightarrow (max, y, z) = \infty; (min, maxSub, T) = (min, minSub, T) + 1 \\
EF &\Rightarrow (max, y, z) = \infty; (min, maxSub, F) = (min, minSub, F) + 1 \\
U &\Rightarrow (max, y, z) = \infty
\end{aligned}$$

Finding a dead-end covering set

Abstract induction proves a formula ϕimpossible by exhaustively enumerating all world abstractions. The worlds are built as a tree called an *extension-abstraction*

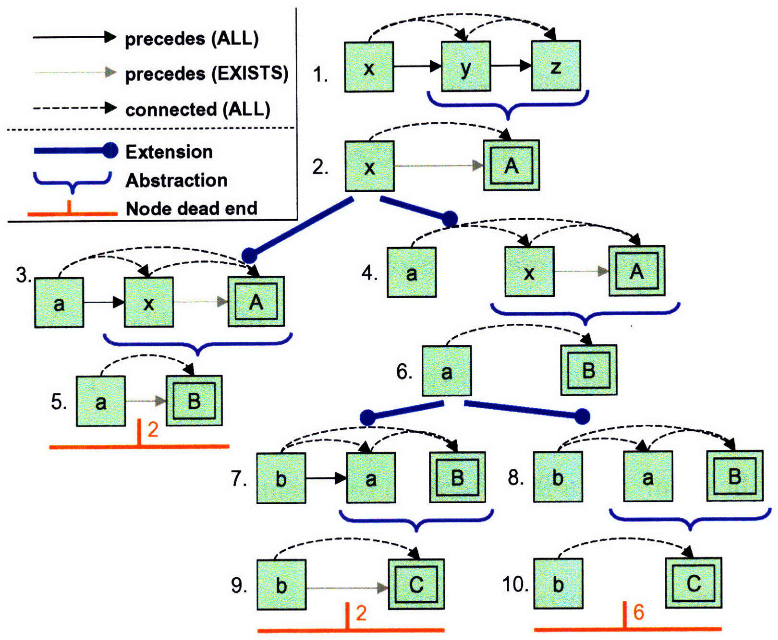

Figure 3-17: A portion of an extension-abstraction tree showing how node 1 becomes a dead-end.

tree. A portion of a tree is shown in Figure 3-17. Nodes in the tree are abstractions. For each tree node α, the *extension children* are all of the extensions of α made by adding a single concrete term and forming relations to each of α's existing objects and wildcards. The *abstraction children* are all of the different mergers that can be made between two objects, an object and a wildcard, or between two wildcards. Only those abstractions that are determinable, i.e. that evaluate to N are considered. If any node ever evaluates to A, then a counter-example has been found, and this is returned as a proof failure.

A node on the extension-abstraction tree is proven a dead-end if all of its extensions are dead-ends, or if one of its abstractions is a dead-end. There are two ways to prove that an abstraction $\alpha$is a dead-end for a formula ϕ. The first and easiest way is to reason about the formua directly to show that no possible extensions to α can ever result in truth. For example, in the marked list example, consider the world $\omega = \{(connected\ x\ y), (connected\ y\ x)\}$. No matter how many objects or wildcards are added to ω, it will always violate the constraint $\forall x \forall y \neg(connected\ x\ y) \vee \neg(connected\ y\ x)$. So, for any abstraction, there is a function called $Impossible(\phi, \alpha)$, which checks for violations of $AtLeast(AllBut())$ quantifiers that would prevent any extension of $\alpha$from ever becoming true.

Because *Impossible* abstractions can be checked and discarded easily, the extension-abstraction tree only includes extensions that are not obvious dead-ends. Furthermore, the tree only includes those extensions that satisfy an immediate need of the formula. That is, for an abstraction α that is not impossible, $\phi(\alpha) = N$ because there is some $AtLeast()$ quantification that is not being satisfied. There is a function called $Extensions(\phi, \alpha)$ which returns a list of pairs $< (z, \{R\}) >$ that could tackle the open quantification. z is a fresh term to represent the new extension object, and R is a set of relations on z to fix the open quantification. By focusing on a specific issue in the formula and ignoring all extensions that are impossible, the number of extensions to a given abstraction is generally small, so that the exhaustive search is generally tractable and fast.

When α is not obviously impossible, the way to prove it a dead-end is to discover loops in the extension-abstraction tree. A loop occurs when an abstraction child of a node P is equal to an ancestor of that node Q. This is shown in Figure 3-17. In this case, P terminates in a valid abstraction for ϕ if and only if Q terminates as well. Thus, there is no benefit in continuing to expand P. P is called a *node dead-end* from Q. This does not necessarily mean that α_P is a dead-end, but any satisfying extension of α_P from P would be found from Q. We also say that P is a node dead-end for any parents of Q, $\bar{Q}$. Let R be an extension parent of P. If all of R's, extensions become loops like this, then R is a node dead end from Q. If R is an abstraction child of a more concrete node R^o, then R^o is also a node dead-end from Q.

By finding loops, the algorithm produces node dead-ends throughout the extension-abstraction tree. A particular abstraction α becomes invalid if there is a sub-tree rooted at node P with $\alpha_P = \alpha$ and all sub-trees of P are node dead-ends from P. Intuitively, this means that all extensions of α must loop forever without ever satisfying the formula ϕ. In this way, the extension-abstraction tree eventually gets fully pruned, there are no more nodes to explore, and the algorithm has found a dead-end covering set.

Let us examine how this works in the marked list example. Call the axiom formula ϕ_A, written in Equation 4.1. The goal is to prove $\exists z \forall x \neg(\mathit{connected}\ x\ z)$. So, the abstract induction theorem prover must show that $\phi_A \wedge \forall z \exists x\ (\mathit{connected}\ x\ z)$ is impossible to satisfy. The tree building algorithm builds all non-impossible instances of size 3. One of these instances is shown as node 1, P_1, in Figure 3-17. This instance evaluates to N, because, among other reasons, the first element does not have an object connected to it. The world is abstracted to the two element abstraction in node P_2. This evaluates to N for the same reason. At this point, the algorithm forms all of the extensions to the abstraction that are not impossible and that satisfy $(\mathit{connected}\ z\ x)$. There are two such extensions, one where $(\mathit{precedes}\ z\ x)$ and one where $\neg(\mathit{precedes}\ z\ x)$. Note that $(\mathit{connected}\ z\ A)$ must evaluate to All. By the transitivity rule of the list definition, if $(\mathit{connected}\ x\ A)$ is All, then if $(\mathit{connected}\ z\ A)$ is not All, the formula ϕ_A becomes impossible. The two nodes P_3 and P_4 merge into abstractions P_5 and P_6 in Figure 3-17. The first abstraction P_5 is exactly the same as node P_2. Therefore, P_5 is a node dead-end from P_2. Now, P_6 expands in two ways to satisfy $\exists z \forall x \neg(\mathit{connected}\ x\ z)$. These two extensions are abstracted into nodes P_9 and P_{10}. P_9 is identical to nodes P_5 and P_2. P_{10} is identical to node P_6. Since they are both dead-ends from P_2, P_6 is a node dead-end from P_2. Thus, the abstraction α_{P_2} and the abstraction α_{P_1} are all dead-ends. In a similar manner, the whole tree is explored, and the formula $\phi_A \wedge \forall z \exists x\ (\mathit{connected}\ x\ z)$ is eventually proven false. No abstraction larger than 3 elements is ever examined in the search.

The mechanics of the search uses a priority queue. Abstractions are extended one at a time, and smaller abstractions are placed at the head of the queue. To conclude the marked list example, the two proof obligations in Equation 3.9 are presented to the abstract induction prover, and they complete in under a minute.

3.4 Conclusion

This chapter has presented the first of the three SSGP systems, Form2Grammar. While the task is the simplest, the techniques used to solve the problem are the same as those in the other two systems. The core technologies of deep simplification and abstract induction are used heavily in Spec2Action and HELPS. More fundamentally, Form2Grammar demonstrates the effectiveness of using examples to reason about a complex logical domain. The examples are produced and solved in a simple, concrete, propositional domain, and the generalization of the example solutions forms the hypothesis to be proved by the higher-level deduction system.

The task of Form2Grammar is to take a quantified formula S and produce a recursive constraining grammar that produces exactly those worlds that satisfy S. Using examples, Form2Grammar produces a hypothesis constraining grammar. To prove the grammar correct, Form2Grammar must show that the left side constraining formulas of this grammar imply the right side breakdown formulas.

To generate the hypothesis, Form2Grammar generates example worlds of S, and it finds a consistent set of breakdowns that work for all examples. These breakdowns are then unified together to produce a finite state machine with the states representing grammar non-terminals and the arcs representing breakdowns. From the finite-state machine, the hypothesis grammar is created. Then, using S as the top constraining formula, constraining sub-formulas are produced for all of the *in* non-terminals of the grammar. These sub-formulas may be defined in terms of a special interface partition. To properly reason about the formulas, Form2Grammar uses the examples to guess a mapping from the interface partition of the left-side formula to combinations of the breakdown partition and the interface partitions of the sub-productions.

Once the constraining formulas are in place and the interface partitions mapped, Form2Grammar produces proof obligations. It must show that the constraining left-side formula implies the right-side production. It first simplifies the right-side production in terms of the breakdown partition. Then, it substitutes pivot relationships for the breakdown sets to simplify the right side even further. Finally, the resulting simple expression is passed to the abstract induction theorem prover. This theorem prover assumes the hypothesis and negates the formula to be proved. It shows that all worlds are impossible by showing that a search for a satisfying counter-example must necessarily loop forever.

Chapter 4

Spec2Action

4.1 Introduction

The second SSGP sub-system is called Spec2Action. It is the first actual program synthesis system that will be examined. It takes the specification of a program as two quantified formulas, a start S and a goal G. The output is a description of the relations that need to change in a world ω to make G hold. This relation change set is called an *action*. This is a simpler problem than full program synthesis with STRIPS actions. A STRIPS action has preconditions, and its effect is generally to change more than one relation. In STRIPS planning, one must pay very close attention to the order of actions, as the side-effect of action A may negate the desired effect of action B (like the Sussman anomaly). Also, due to pre-conditions, a given STRIPS action can only be run under special circumstances.

Because each relation is changed individually and independently, there are no constraints on the order in which the individual relations are changed. Thus, in Spec2Action, the programs synthesized have no sense of timing. The goal is to take two formulas S and G and produce a set of relations. More broadly, Spec2Action is designed to be a subsystem for the more general HELPS application. When given a program synthesis problem, Spec2Action determines *what* to do, what facts about the world need to change to satisfy the goal. The higher-level HELPS must then determine *how* to use STRIPS actions to set the relations established by Spec2Action.

The exercising test cases to guide Spec2Action development are operations on data structures. These are problems where determining the relations to change is non-trivial, so they serve as good tests of the system's algorithms and representations. Figuring out the relations to change from the logical spec requires uncovering the recursive structure of worlds satisfying the goal formula, uncovering specific roles that are filled by particular objects, and figuring out how the worlds change with respect to those roles.

For example, consider removing a marked element from a linked list. The start formula S is the marked list specification studied extensively in Chapter 3. Repeated for clarity, it is:

$$\begin{aligned}
& \forall x U y \; (\textit{precedes } x \; y) \\
\wedge \;\; & \forall x \forall y \; (\textit{connected } x \; y) \;\Leftrightarrow \\
& \quad (\textit{precedes } x \; y) \vee (\exists z \; (\textit{connected } x \; z) \wedge (\textit{connected } z \; y)) \\
\wedge \;\; & \forall x \forall y \; (\textit{connected } x \; y) \;\Leftrightarrow \neg(\textit{connected } y \; x) \\
\wedge \;\; & D x \; (\textit{marked } x)
\end{aligned} \tag{4.1}$$

The goal is to change the relations so that all of the elements except the marked one form a list.

$$\begin{aligned}
& \forall x \; \neg(\textit{marked } x) \Rightarrow U y \; \neg(\textit{marked } y) \wedge (\textit{precedes } x \; y) \\
\wedge \;\; & \forall x \forall y \; (\neg(\textit{marked } x) \wedge \neg(\textit{marked } y)) \Rightarrow \\
& \quad ((\textit{connected } x \; y) \Leftrightarrow \\
& \quad\quad (\textit{precedes } x \; y) \vee (\exists z \; \neg(\textit{marked } z) \wedge (\textit{connected } x \; z) \wedge (\textit{connected } z \; y)) \\
\wedge \;\; & \forall x \forall y \; (\neg(\textit{marked } x) \wedge \neg(\textit{marked } y)) \Rightarrow (\textit{connected } x \; y) \Leftrightarrow \neg(\textit{connected } y \; x)
\end{aligned} \tag{4.2}$$

There are many solutions to this problem, including obliterating all existing rela-

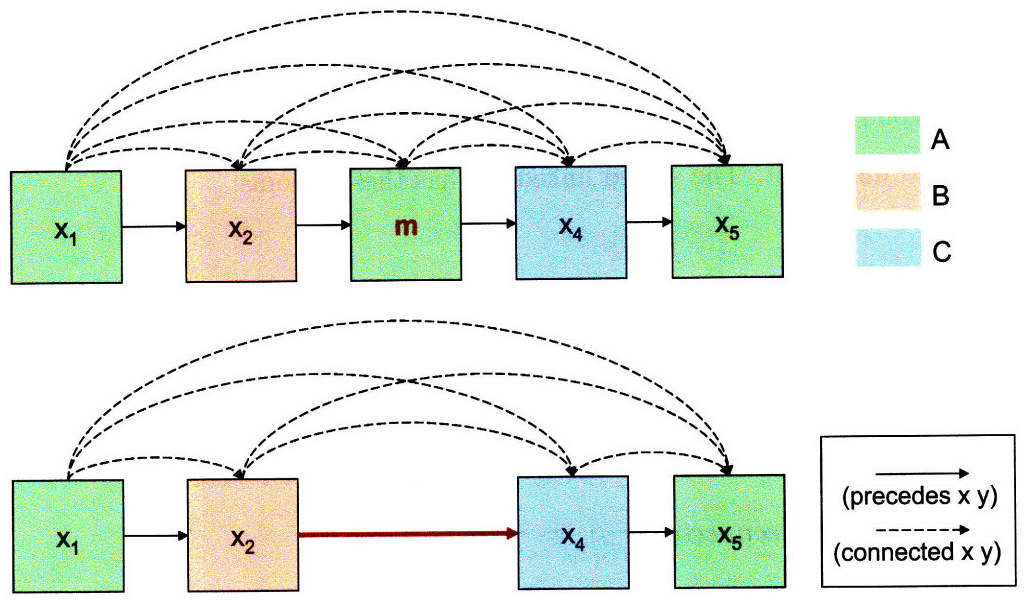

Figure 4-1: The relations to change when removing the marked element from a list

tions, and then resetting them in the desired configuration. However, Spec2Action's goal, more specifically, is to change as few relations as possible to make the goal true. A general solution to remove the marked element is shown in Figure 4-1. The two elements B and C on either side of the marked element are picked out, and the relation (*precedes* B C) is set between them.

In particular, Spec2Action only produces a certain type of relation change set. It finds a partition of the world, and the relations to change are *set relations* between the sets of the partition. A set relation $R(A_1, A_2, \ldots, A_n)$ defines the set of relations $R(a_1, \ldots, a_n)$ for $a_i \in A_i$. For a program of set relations $\{R\}$ to be correct, every world ω satisfying the start formula S must break up into a partition $\{A_1, \ldots, A_n\}$ such that R applied to this partition changes the world to satisfy the goal formula. The fixed set partition $\{A_1, \ldots, A_n\}$ determines the roles of the different objects in a program.

For example, in the marked list case, the partition is $\{A, B, C\}$, where B is the singleton set of the element before the marked element, C is the singleton set of the element after the marked element, and A is the rest. B and C may be null if the marked element is at the beginning or end of the list respectively. The changed relation is $\{(\textit{precedes}\ B\ C)\}$.

As a second example, consider the problem of inserting an orderable element a into a sorted linked list. The sorted linked list has these axioms:

$$\begin{aligned}
& \forall x U y\ (\textit{precedes}\ x\ y) \\
\wedge\ & \forall x \forall y\ (\textit{connected}\ x\ y) \Leftrightarrow \\
& \quad (\textit{precedes}\ x\ y) \vee (\exists z\ (\textit{connected}\ x\ z) \wedge (\textit{connected}\ z\ y)) \\
\wedge\ & \forall x \forall y\ (\textit{connected}\ x\ y) \Leftrightarrow \neg(\textit{connected}\ y\ x) \\
\wedge\ & \forall x \forall y \forall z\ (\leq x\ y) \wedge (\leq\ y\ z) \Rightarrow (\leq x\ z) \\
\wedge\ & \forall x \forall y\ (\leq\ x\ y) \Leftrightarrow \neg(\leq\ y\ x) \\
\wedge\ & \forall x \forall y\ (\leq\ x\ y) \Leftrightarrow (\textit{connected}\ x\ y)
\end{aligned}$$

The set partition is $\{A, B, C, D\}$ where B and C are singleton (or null) sets representing the elements of the total ordering just before a and just after a respectively. The relations to change are:

$$\begin{aligned}
&\{(\textit{precedes}\ B\ a), (\textit{precedes}\ a\ C), \neg(\textit{precedes}\ B\ C), (\textit{connected}\ A\ a), \\
&(\textit{connected}\ B\ a), (\textit{connected}\ a\ C), (\textit{connected}\ a\ D), \neg(\textit{precedes}\ a\ B), \\
&\neg(\textit{precedes}\ a\ A), \neg(\textit{precedes}\ A\ a), \neg(\textit{connected}\ a\ A), \neg(\textit{connected}\ a\ B), \\
&\neg(\textit{precedes}\ C\ a), \neg(\textit{precedes}\ a\ D), \neg(\textit{precedes}\ D\ a), \neg(\textit{connected}\ C\ a), \\
&\neg(\textit{connected}\ D\ a)\}
\end{aligned}$$

Since a is being inserted, all of its relationships with each of $\{A, B, C, D\}$ must be established. The sorted list brings up an issue. Only certain types of relations are changeable. The *precedes* and *connected* relations are structural relations of the list, and can be changed at will. However, the $\leq$ relation is a fundamental property of the list elements and must not be changed.

The types of programs that can be represented well in this framework is limited. Section 4.3 will argue why this program representation is a fundamental one, and why the program synthesis performed by Spec2Action is an important foundation step for more complex forms of synthesis. First, Section 4.2 will describe the set of challenge problems that guide the development. Section 4.4 then describes the detailed architecture and implementation of Spec2Action.

The SSGP design philosophy of combining inductive and deductive reasoning to solve problems guides this architecture. Given start and goal formulas S and G, Spec2Action generates sample starting states. It then solves these starting states in the most general and consistent way. Then, it forms a hypothesis program consisting of the roles and the relations to change. It then absorbs these changes into the start formula and simplifies the goal formula given the changed start. The relation changes are then pulled backward from this simplified goal. The resulting formula is then a set of conditions that the partition must satisfy. At this point, the goal is to show that any world satisfying the start formula S can be broken down into a partition, such that the simplified and transformed goal holds true. Finally, the role partition is built up from the grammar of S discovered by Form2Grammar. This algorithm is called the set partition builder. If all goes well in this stage, then the partition exists and the hypothesized program is correct and well defined.

4.2 Challenge Problems

This section describes the test problems have been used to guide development. All of these problems are simple operations on data structures. As mentioned above, these operations are appealing, because they cover a variety of non-trivial logical situations, and they exercise the expressive power of the Spec2Action representations and algorithms. Also, data structures are natural to conceive of as networks of objects and relations. Operations on data structures are natural problems in changing the relations of the network.

The problems to be considered are:

1. Inserting an item into a linked list
2. Inserting an item into a doubly linked list
3. Removing an item from a linked list
4. Inserting an item into a binary search tree
5. Removing an item from a binary search tree
6. Inserting an item into a circularly linked list
7. Removing an item from a circularly linked list
8. Inserting an item into a min heap
9. Removing an item from a min heap
10. Inserting an item into a sorted list
11. Moving the minimum item of an unsorted list to the front

This list of challenges provides a wide coverage of the types of complications that arise in specifying the relations to change in a program synthesis problem. Some of the problems, such as problems 1 and 5, consist entirely of adding relations to a world. An item is added to a linked list at the end and to a binary search tree at the bottom. Other problems, like 3 and 6, require negating relations in the existing structures of the directed linked list and the circularly linked list respectively.

The most complex problems are 5, 8, and 9. In fact, in these three cases, there does not exist a general solution template made of set relations over a set partition. That is, the Spec2Action program representation cannot represent a solution to these problems, and the program synthesis must always fail. However, in each of these cases, the natural solution involves a recursion down the branches of a tree. For example, removing an item from a binary search tree involves merging two trees together,

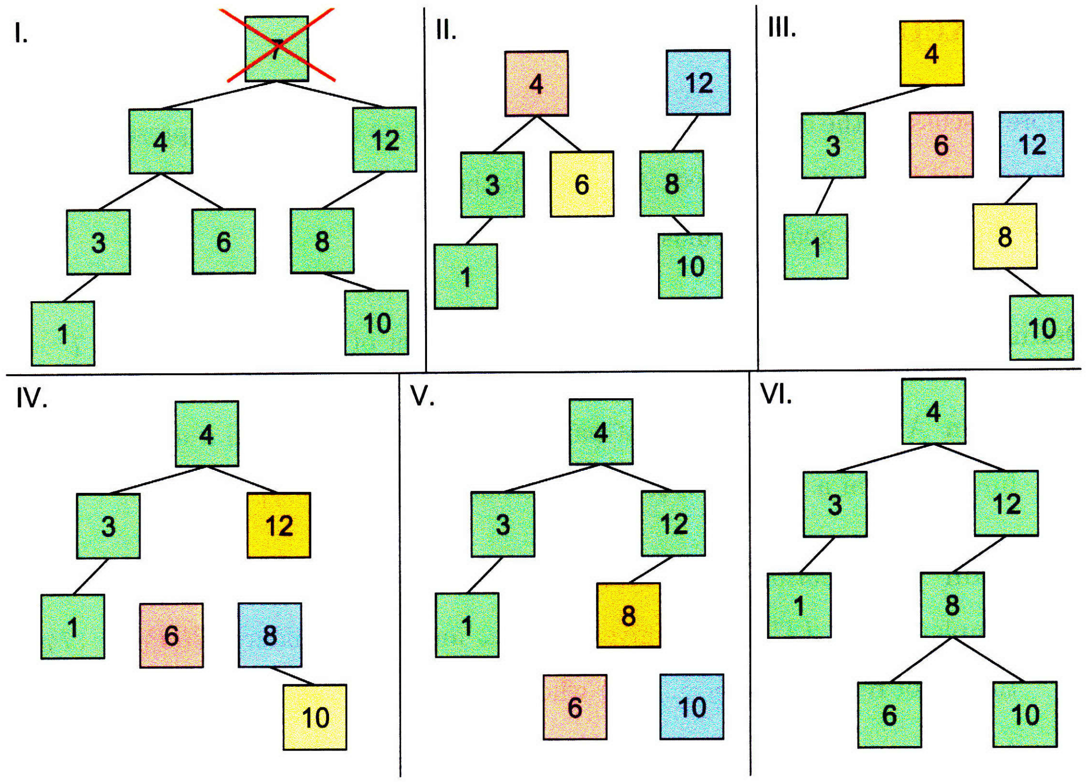

Figure 4-2: The steps to remove an element from a binary search tree. The recursive merging of sub-trees is too complex to represent as an action template.

which requires recursively merging two sub-trees (see Figure 4-2). The functionality of the recursive step is to take two trees T_1 and T_2 such that $(\leq T_1\ T_2)$ and produce three smaller trees S_1, S_2, and S_3 and an element r such that $(\leq S_1\ S_2)$, $(\leq S_2\ S_3)$, $(left\ S_1\ r)$, $(right\ S_2\ r)$, and $(right\ S_3\ r)$.

Thus, while the algorithm as a whole does not fit into an action template, the key recursive step does fit into an action template. Similarly, the key recursive steps of programs 8 and 9 also fit into action templates. Thus, all of the problems on the list have action representations as core parts of the solutions. So, while Spec2Action does not have the full power to solve any kind of programming problem, its capabilities serve as an important foundation step to solving more complex problems.

4.3 Representations

Before discussing the architecture of Spec2Action and its results, the representation of actions must be precisely defined. An action is the output that Spec2Action produces, given the start and goal formulas S and G.

Definition A *set partition template* is an abstract set of n set terms $T^1, \ldots, T^n$.

Definition A *set builder* σ for a set partition template $\{T^i\}_n$ and a universe Ω is a function from a world $\omega \in \Omega$ to a partition of ω $\{\sigma(\omega, T^i)\}$.

Definition A *set relation* $R(P_1, \ldots, P_k)$ is defined on set templates of a set partition template, $P_j \in \{T^i\}$. For a given world ω and a set builder σ $R(P_1, \ldots, P_k)$ is the set of relations $R(o_1, \ldots, o_k)$ where $o_j \in \sigma(\omega, P_j)$ and such that $o_i \neq o_j$ for $i \neq j$.

Definition An *action template* is a set partition template $T^1, \ldots, T^n$ and a set of set relations $\{R(P_1, \ldots, P_k)\}$, where $P_j \in \{T^i\}$.

Definition An *action* consists of an action template along with a set builder for the set partition template.

The goal of Spec2Action is to take a start and goal quantified formula S and G and produce an action. There are thus two things Spec2Action must do: produce the action template, and then produce the set builder on this template. The set builder is defined recursively over the grammar for S as produced by Form2Grammar.

That is, every *in* non-terminal NT is assumed to have a set builder $\sigma_{NT}(\omega, T^i) \subset \omega$. Let a grammar decomposition look like the following:

$$NT ::= piv; \{\{A_{ij}\} \rightarrow NT_i\}; \chi(piv, A_{ij})$$

For a world ω, let $\{piv, \ldots, \omega_i = \bigcup_j \omega_{ij}, \ldots\}$ be a correct parse for this decomposition. Then, $\sigma_{NT}(\omega, T^i)$ is defined to be a union of sets from $\{piv, \ldots, \omega_{ij} \cap$

$\sigma_{NT_i}(\omega_i), \ldots\}$. That is, the set partitions are built up from intersections of set partitions of sub parses with the parse components. The function must be defined so that $\{\sigma_{NT}(\omega, T^i)\}$ is a disjoint partition of ω.

Definition A *recursive set builder* is a set builder defined over a grammar $\mathcal{G}$ so that there is a set builder for each non-terminal, defined as the union of intersections of parse components and the output of the sub production set builders.

Thus, the output of Spec2Action is an action template and a recursive set builder defined over the grammar of the start formula $\mathcal{G}(S)$.

Let us consider a simple example, insertion into a linked list. The start formula S is:

$$\begin{aligned}
& \forall x U y \ (\mathit{precedes}\ x\ y) \\
\wedge \quad & \forall x \forall y \ (\mathit{connected}\ x\ y) \ \Leftrightarrow \\
& \qquad (\mathit{precedes}\ x\ y) \vee (\exists z\ (\mathit{connected}\ x\ z) \wedge (\mathit{connected}\ z\ y)) \\
\wedge \quad & \forall x \forall y \ (\mathit{connected}\ x\ y) \ \Leftrightarrow \neg(\mathit{connected}\ y\ x) \\
\wedge \quad & \forall x \ \neg(\mathit{precedes}\ a\ x) \wedge \neg(\mathit{precedes}\ x\ a) \wedge \neg(\mathit{connected}\ a\ x) \wedge \neg(\mathit{connected}\ x\ a)
\end{aligned}$$

The goal formula G is:

$$\begin{aligned}
& \forall x U y \ (\mathit{precedes}\ x\ y) \\
\wedge \quad & \forall x \forall y \ (\mathit{connected}\ x\ y) \ \Leftrightarrow \\
& \qquad (\mathit{precedes}\ x\ y) \vee (\exists z\ (\mathit{connected}\ x\ z) \wedge (\mathit{connected}\ z\ y)) \\
\wedge \quad & \forall x \forall y \ (\mathit{connected}\ x\ y) \ \Leftrightarrow \neg(\mathit{connected}\ y\ x)
\end{aligned}$$

Since the marked element a is free in S and not free in G, it is assumed to be part of the final list and not part of the starting list. For the example solution, the set partition template is $\{A, B, C\}$ and the action template is

$\{(\textit{precedes}\ A\ B), (\textit{connected}\ A\ B), (\textit{connected}\ A\ C)\}$

The program inserts the element a at the front of the list. So, A consists of the singleton set $\{a\}$. B consists of the first element of the starting list, and C is the rest of the starting list. The grammar for the starting formula is the following:

$$\begin{aligned}
M_{in} &::= p; \{P_1 \rightarrow L_{in}\}; \neg(\textit{precedes}\ p\ P_1); \neg(\textit{precedes}\ P_1\ p); \neg(\textit{connected}\ p\ P_1);\\
&\qquad \neg(\textit{connected}\ P_1\ p)\\
L_{in} &::= L_o \mid \epsilon\\
L_o &::= p; \{\{P_1, P_2\} \rightarrow L_{in}\}; \neg(\textit{marked}\ p); (\textit{precedes}\ p\ P_1); (\textit{connected}\ p\ P_1);\\
&\qquad (\textit{connected}\ p\ P_2); \neg(\textit{precedes}\ P_1\ p); \neg(\textit{connected}\ P_1\ p); \neg(\textit{precedes}\ p\ P_2);\\
&\qquad \neg(\textit{precedes}\ P_2\ p); \neg(\textit{connected}\ P_2\ p)
\end{aligned}$$

In M_{in}, the pivot is the element to be inserted a. L_{in} represents the existing list. The pivot is the first element of the list, P_1 is the second element (or null), and P_2 is the rest.

The recursive set builder is:

$$\begin{aligned}
\sigma_{M_{in}} &= \{A = \{p\}, B = \sigma_{L_{in}}(P_1).B, C = \sigma_{L_{in}}(P_1).C\}\\
\sigma_{L_{in}} &= \textbf{if}\ \epsilon\ \textbf{then}\ \{A = \{\}, B = \{\}, C = \{\}\}\\
&\qquad \textbf{else if}\ L_o\ \{A = \{\}, B = \{p\}, C = \sigma_{L_{in}}(\{P_1, P_2\}).B\ \cup\ \sigma_{L_{in}}(\{P_1, P_2\}).C\}
\end{aligned}$$

That is, B, the first element of the list consists of the the pivot point p. C, the rest of the list, consists of the head of the sub-list unioned with the rest of the sub-list. The build-up is shown in Figure 4-3. The next section will show how the action template is created and how the recursive set builder is defined so that the action template combined with the recursive set builder, applied to a world satisfying the start formula, will satisfy the goal formula.

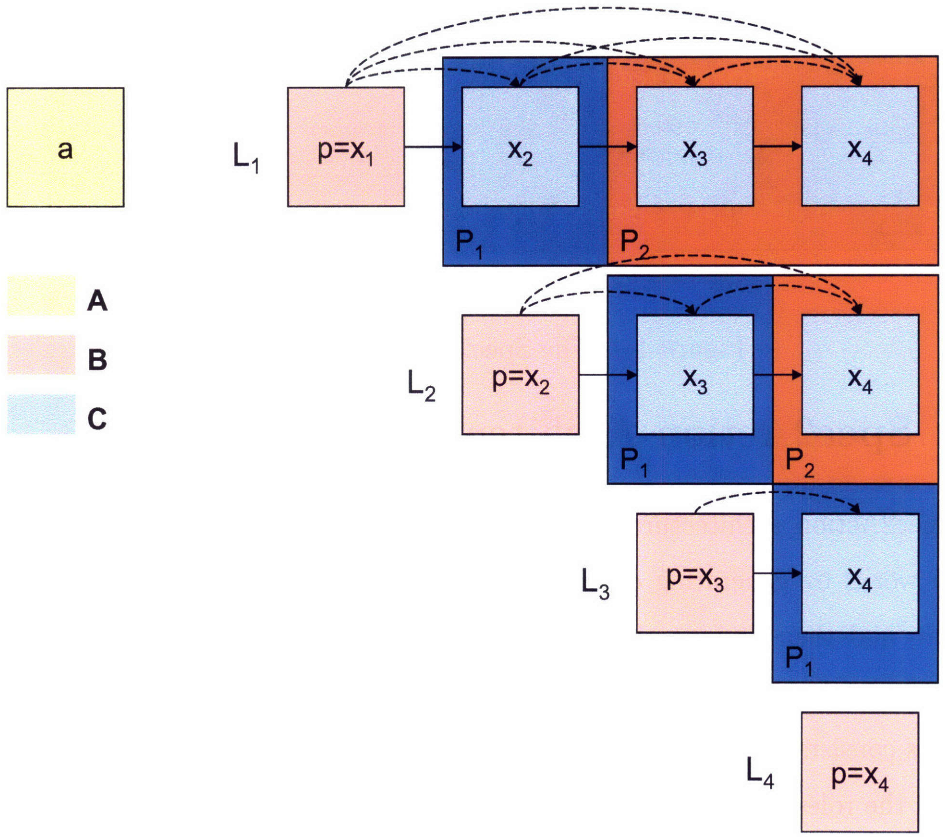

Figure 4-3: The recursive set build-up for the action template partition $\{A, B, C\}$ from the grammar produced by Form2Grammar.

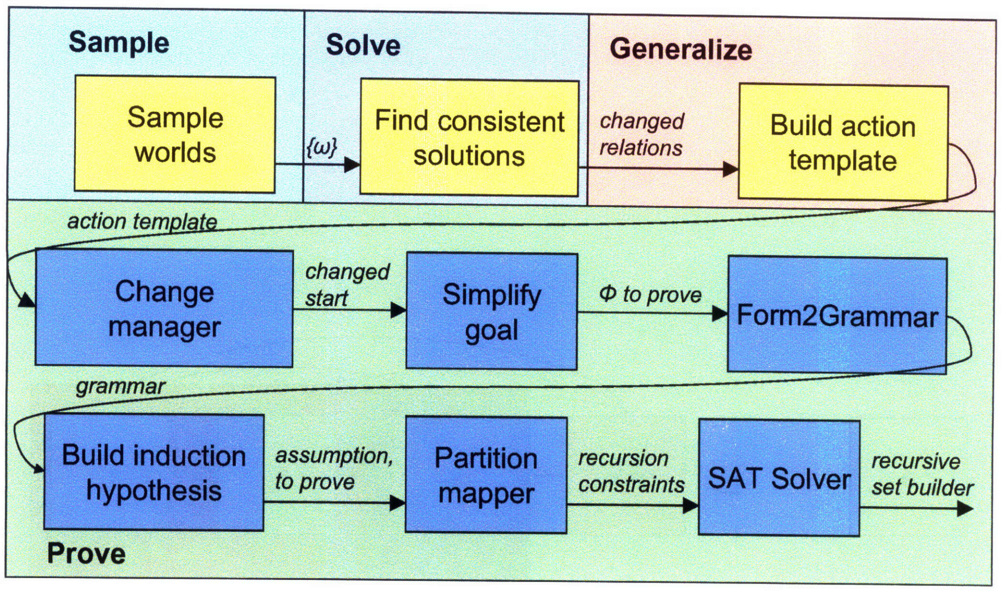

Figure 4-4: The Spec2Action Architecture

4.4 Spec2Action Architecture

The Spec2Action architecture is shown in Figure 4-4. As in Form2Grammar, the system works by generating examples, solving the examples, generalizing from the solution, and then proving the solution correct. For the example set, it generates a consistent, simple set of solutions for all examples. The generalization of these solutions consists of the action template, a set partition of roles, and a set relations between the roles.

To prove this generalization correct, there is a module, the Change Manager, that propagates the action through the start formula, producing a formula that is true after the action has been performed. Then, the goal formula is simplified relative to this transformed start. This provides a set of conditions on the set partition needed to satisfy the goal formula. Finally, the Change Manager propagates the action backward by reversing the effects of the action on the partition conditions. This produces a set of conditions on the partition roles that must be true at the start of the program.

At this point, the goal of the system is to show that the partition satisfying the transformed goal conditions exists. This satisfying partition is built up by structural induction over the grammar of the starting formula $\mathcal{G}(S)$ by the Partition Mapper algorithm. For a given non-terminal NT, it assumes that the role conditions are true for the sub-production non-terminals. Then, the Partition Mapper generates a propositional formula that contains all of the constraints on the recursive set builder to make the goal conditions for NT true. This propositional formula is tested with a SAT solver, and a correct solution fully defines the recursive set builder. Now, Spec2Action has an action template and a recursive set builder, and it is done.

The functioning of these components will now be examined in detail. To explain the system, there will be one example problem throughout the exposition: removing an element from a linked list. The start and goal formulas are given in Equations 4.1 and 4.2 respectively. This example has the marked linked list as the start formula, which was studied in depth in Chapter 3. It has the grammar given by 3.5.

4.4.1 Solving Examples Simply and Consistently

To generate a wide set of examples of the start formula, Spec2Action uses the same example generator as discussed in Section ?? for Form2Grammar. In the case of removing the marked element, the same examples that were generated for the marked linked list can be used.

The simplest solution is judged to be the one with the smallest set partition and the fewest set relation changes within it. Given a set of examples $\omega_1, \ldots, \omega_n$, a maximum partition size d and a maximum number of relation changes r, Form2Grammar tries to find an action template and an assignment for the objects $o \in \omega_i$ to a partition set A_j such that the goal formula holds in the transformed world.

All of these constraints are put into a propositional formula, which is then solved by a SAT solver. Form2Grammar increases d until a particular action template works. Then, it uses binary search on the maximum relation number r to find the absolute simplest action template that is possible for the examples. The Spec2Action system currently uses the zchaff SAT solver.

Consider a simple example. Consider the five-element marked list in Figure 4-2. The middle element needs to be deleted. This can be accomplished by partitioning the list into three sets A, B, and C with the set relation change (*precedes* B C). There are several types of proposition that go into the action template constraint formula. First, there is the instantiation of the goal formula (Equation 4.2) over the elements $e_1, \ldots, e_5$. Furthermore, each element is constrained to lie in exactly one of the partition sets: e.g.

$$\begin{aligned} & (A\ e_1) \vee (B\ e_1) \vee (C\ e_1) \\ \wedge\ & \neg(A\ e_1) \vee \neg(B\ e_1) \\ \wedge\ & \neg(A\ e_1) \vee \neg(C\ e_1) \\ \wedge\ & \neg(B\ e_1) \vee \neg(C\ e_1) \end{aligned}$$

Then, each set relation has two propositions, only one of which can be true: (*set* (*precedes* B C)) and (*unset* (*precedes* B C)). If both of these propositions are set to false, then the (*precedes* B C) set relation is not changed. For (*set* (*precedes* B C)) to be a correct set relation for two elements e and f with (B e) and (C f), then it must be the case that $\neg$(*precedes* e f). Similarly, to apply (*unset* (*precedes* B C)), then (*precedes* e f) must be true. That is, a set relation change must actually change all of the relations for which it is applied. This restriction is captured in constraints such as:

$$\neg(B\ e_1) \vee \neg(C\ e_2) \vee \neg(\textit{set}\ (\textit{precedes}\ B\ C))$$

Because (*precedes* e_1 e_2), if (B e_1) and (C e_2), then (*precedes* B C) cannot change to true. In a similar way, there is a proposition

$$\neg(B\ e_1) \vee \neg(C\ e_2) \vee \neg(\textit{unset}\ (\textit{precedes}\ C\ B))$$

(*precedes* e_2 e_1) cannot be changed to false, because it already is false.

For a relation such as (*precedes* e_1 e_2) to be true after the action templates have run, then (*unset* (*precedes* B C)) must not be the set relation when (B e_1) and

$(C\ e_2)$. The rule here is:

$$\neg(B\ e_1) \vee \neg(C\ e_2) \vee ((\textit{unset}\ (\textit{precedes}\ B\ C)) \Leftrightarrow \neg(\textit{precedes}\ e_1\ e_2))$$

Finally, there are propositonal rules to make the number of relation changes at most r. All of these constraints are put into a large formula, which is then solved by SAT. The satisfied assignment for the smallest r for the smallest d becomes the hypothesized solution. In the case of the list removal, the three-set partition $\{A, B, C\}$ is hypothesized with the single changed set relation (*precedes* B C).

4.4.2 Propagating action templates forward

With the hypothesized action template $(\{T^i\}, R)$, the problem is now to show that the action template works for all start and goal formulas S and G. To do this, we must show that a partition $\{T^i\}$ exists such that, after the set relation changes R are run, the resulting world satisfies the goal G. To do this, the start formula S is first broken down over the partition $\{T^i\}$. Then, it is assumed that the changed set relations R are false, F, in the partitioned formula $S : \{T^i\}$. This simplifies $S : \{T^i\}$ slightly. Finally, the relations R are added as universally quantified formulas over $\{T^i\}$. Consider list removal as an example.

Look at the formula $\forall x U y$ (*precedes* x y). This is broken up over the partition

$\{A, B, C\}$ as

$$\begin{aligned}
&\forall x : A && (Uy : A\ (precedes\ x\ y) \wedge \forall y : \{B, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : B\ (precedes\ x\ y) \wedge \forall y : \{A, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : C\ (precedes\ x\ y) \wedge \forall y : \{A, B\}\ \neg(precedes\ x\ y)) \\
\wedge\ &\forall x : B && (Uy : A\ (precedes\ x\ y) \wedge \forall y : \{B, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : B\ (precedes\ x\ y) \wedge \forall y : \{A, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : C\ (precedes\ x\ y) \wedge \forall y : \{A, B\}\ \neg(precedes\ x\ y)) \\
\wedge\ &\forall x : C && (Uy : A\ (precedes\ x\ y) \wedge \forall y : \{B, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : B\ (precedes\ x\ y) \wedge \forall y : \{A, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : C\ (precedes\ x\ y) \wedge \forall y : \{A, B\}\ \neg(precedes\ x\ y))
\end{aligned}$$

When the fact that $\neg(precedes\ B\ C)$ is applied before the action is run, this simplifies to:

$$\begin{aligned}
&\forall x : A && (Uy : A\ (precedes\ x\ y) \wedge \forall y : \{B, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : B\ (precedes\ x\ y) \wedge \forall y : \{A, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : C\ (precedes\ x\ y) \wedge \forall y : \{A, B\}\ \neg(precedes\ x\ y)) \\
\wedge\ &\forall x : B && (Uy : A\ (precedes\ x\ y) \wedge \forall y : B\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : B\ (precedes\ x\ y) \wedge \forall y : A\ \neg(precedes\ x\ y)) \\
\wedge\ &\forall x : C && (Uy : A\ (precedes\ x\ y) \wedge \forall y : \{B, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : B\ (precedes\ x\ y) \wedge \forall y : \{A, C\}\ \neg(precedes\ x\ y)) \\
& && \vee(Uy : C\ (precedes\ x\ y) \wedge \forall y : \{A, B\}\ \neg(precedes\ x\ y))
\end{aligned}$$

All instances of expressions of the form $(precedes\ B\ C)$ are set to F. All such formulas in 4.2 are simplified to produce the transformed start formula S'. The resulting formula now has no expressions $(precedes\ B\ C)$. So, when the action template is applied, none of the other expressions or relations are affected. The above simplified formula for $\forall x U y$ (*precedes* x y) over $\{A, B, C\}$ is true before the action is run and after it. To complete the transformation, S' is augmented with the set relation formula $\forall x : B \forall y : C$ (*precedes* x y). S' is true both before and after the action run. However, $S' \wedge \forall x : B \forall y : C$ (*precedes* x y) is only true after the action run.

The transformation of a formula S through an action template $(\{T^i\}, R)$ is denoted $\tau_R(S)$.

4.4.3 Simplification

Once the start formula has been transformed by the action template $(\{T^i\}, R)$ to create $\tau_R(S)$, the goal formula $G : \{T^i\}$ is partitioned over T^i and simplified, given the transformed start. The simplification consists of shallow simplification followed by deep simplification, as described in Section 3.3.7.

Usually this simplification simplifies the goal considerably. In the marked list example, the simplified goal is denoted it by G'. It is, with some consolidation, roughly:

$$\begin{array}{lll}
& & \forall x : B\ \forall y : C\ (\neg(\textit{marked } x) \wedge \neg(\textit{marked } y)) \Rightarrow (\textit{connected } x\ y) \\
\wedge & & \forall x : \{A,B,C\} \forall y : \{A,B,C\}\ (\neg(\textit{marked } x) \wedge \neg(\textit{marked } y) \wedge \\
& & \quad (\textit{connected } x\ y) \wedge \neg(\textit{precedes } x\ y)) \Rightarrow \\
& & \qquad \exists z : \{A,B,C\}\ \neg(\textit{marked } z) \wedge (\textit{connected } x\ z) \wedge (\textit{connected } z\ y) \\
\wedge & & \textit{AllBut}(1)\ x : C\ (\textit{marked } x) \\
& \vee & \forall x : B\ (\textit{marked } x) \\
\wedge & & \forall x : C\ (\textit{marked } x) \\
& \vee & \forall x : B\ \forall y : \{A,B\}\ (\neg(\textit{marked } x) \wedge \neg(\textit{marked } y)) \Rightarrow \neg(\textit{precedes } x\ y)
\end{array}$$

The goal is to show that there exists a partition $\{A, B, C\}$ of $\tau_R(S)$ such that G' holds. This is equivalent to showing that there exists a partition $\{A, B, C\}$ of S

such that $G' \wedge \forall x : B \forall y : C \neg(\mathit{precedes}\ x\ y)$ holds. The proof of the existence of this partition is made by reasoning over the grammar for S, $\mathcal{G}(S)$.

4.4.4 Recursive Set Builder

Now, the problem is to construct a partition $\{T^k\}$ of a particular formula S such that a set of conditions G' holds over $\{T^k\}$. This is proven by induction over the grammar $\mathcal{G}(S)$. Consider a particular non-terminal rule with a pivot *piv*, sub-productions NT_i and cross relations $\chi(\mathit{piv}, A_{ij})$:

$$NT ::= \mathit{piv}; \{\{A_{ij}\} \rightarrow NT_i\}; \chi(\mathit{piv}, A_{ij})$$

Assume by induction that the sub-productions NT_i correctly break up into a partition $\{T^k\}$ over the subset $\{A_{ij}\}$ satisfying some portion of the goal formula G' which depends on the non-terminal type of NT_i. Using the information from the induction hypothesis, as well as the pivot cross relations χ, Spec2Action constructs the satisfying partition $\{T^i\}$ out of all of these sub-partitions.

The key general algorithm to make this possible is the Partition Mapper, which will be described in the next section. Given an initial formula S quantified over a partition $\{T^i\}$ and a desired formula G quantified over a second partiton $\{U^j\}$, the partition mapper finds a mapping $\rho : \{T^i\} \rightarrow \{U^j\}$ such that $S \Rightarrow G : \{\rho^{-1}(U^j)\}$. That is, S and $\{T^i\}$ can be used to construct a partition $\{U^j\}$ such that $G : \{U^j\}$.

Consider a simple example from inserting an element into an ordinary linked list, which has a simpler grammar and simpler sufficient conditions than the program of removing an element from a linked list. The grammar for the unmarked list is (omitting negative pivot relations):

$$\begin{aligned} L_{in} &::= L_S | \epsilon \\ L_S &::= p;\ \{\{P_1, P_2\} \rightarrow L_{in}\}; (\mathit{precedes}\ p\ P_1), (\mathit{connected}\ p\ P_1), \\ &\qquad (\mathit{connected}\ p\ P_2) \end{aligned} \tag{4.3}$$

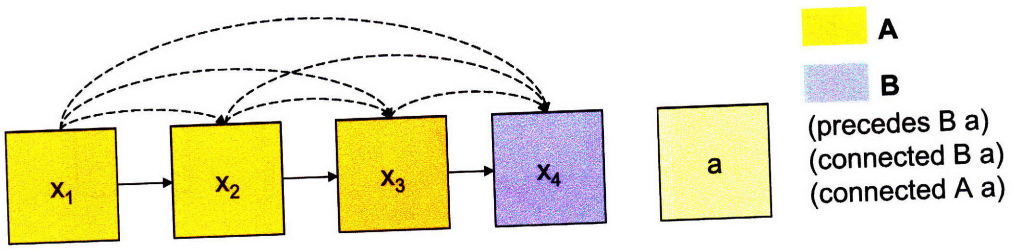

Figure 4-5: Sets A and B for inserting an element a at the end of a list

The goal is to insert a single element a at the end of this list. The action template to make this happen is:

$$\{A, B\}; (connected\ A\ a),\ (precedes\ B\ a),\ (connected\ B\ a)$$

Using the simplification method of the previous section, the simplified conditions for A and B are:

$$\begin{aligned} \nu : \{A, B\} = \quad & \forall x : A\ \exists y : \{A, B\}\ (connected\ x\ y) \\ \wedge \quad & \forall x : B\ \forall y : \{A, B\}\ \neg(precedes\ x\ y) \end{aligned} \tag{4.4}$$

That is, every element of A is connected to something later on in the list and there is no element after the singleton set B. This is shown in Figure 4-5. How would one prove the existence of the partition $\{A, B\}$ for all lists, and would one construct the partition? Both questions are answered by induction.

The base case is when $L_{in} = \epsilon$. This clearly breaks into A and B that satisfy the conditions. Then, to prove L_S breaks down properly, assume that the sub-production L_{in} breaks down into A' and B' satisfying 4.4. Call this induction formula $\nu : \{A', B'\}$ and call the pivot cross-relation production formula $\chi : \{p, P_1, P_2\}$. So, we have to construct A and B satisfying $\nu : \{A, B\}$ out of the overlapped partition $\{\{p\}, P_1 \cap A', P_2 \cap A', P_1 \cap B', P_2 \cap B'\}$ and given

$\nu : \{A' \cap (P_1 \cup P_2), B' \cap (P_1 \cup P_2)\} \wedge \chi : \{p, P_1 \cap \{A', B'\}, P_2 \cap \{A', B'\}\}$.

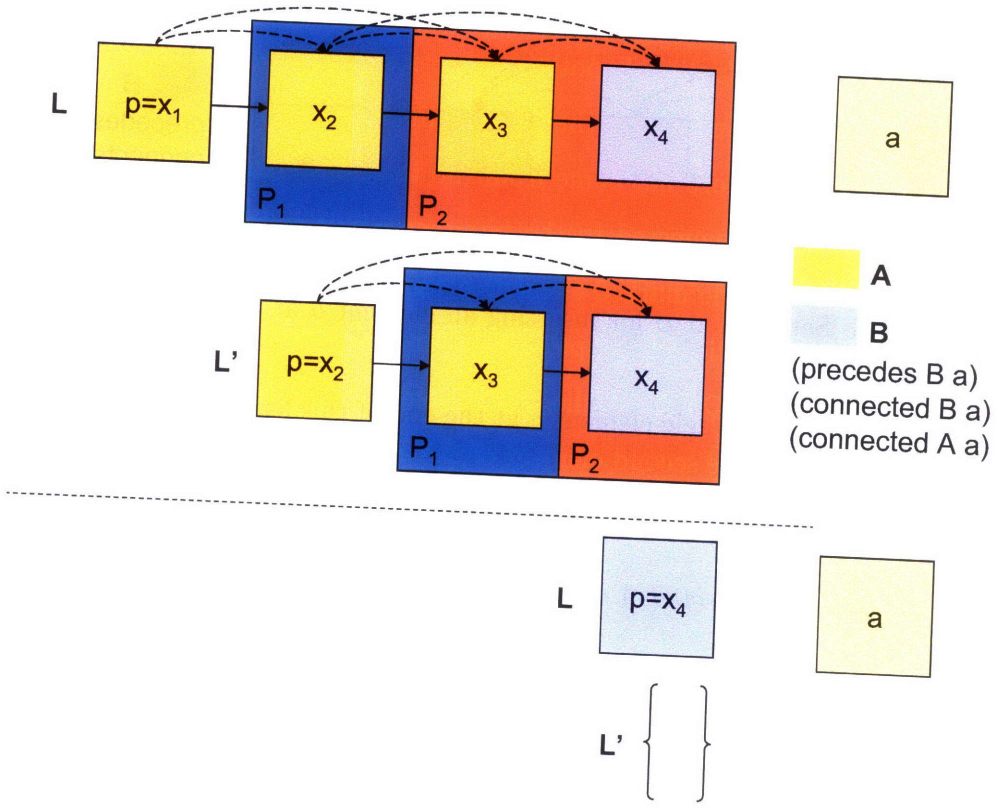

Figure 4-6: Sets A and B built recursively from the grammar breakdown $\{p, P_1, P_2\}$ and A' and B' of the sub-list L'. There are two cases for when $L'$is or is not ϵ.

When A' and $B'$are empty, $\nu : \{A, B\}$ is true when $B = \{p\}$ and $A = \{\}$. Otherwise, it is true when $B = (B' \cap P_1) \cup (B' \cap P_2)$ and $A = \{p\} \cup (A' \cap P_1) \cup (A' \cap P_2)$. That is, if $A'$and $B'$are null, then B, the new end of the list is the pivot point p. If A' and $B'$are not null, then A consists of the old A' and the pivot, while B consists of the old B'. These combinations are shown in Figure 4-6.

Now, for a general case, there is a grammar decomposition production rule:

$$NT ::= p; \{\{A_{ij}\} \rightarrow NT_i\}; \chi(p; A_{ij})$$

To show the existence of a partition $\{T^k\}$ satisfying a formula ν_{NT}, Spec2Action assumes that some version of the formula ν_{NT_i} holds over a partition $\{U_i^k\}$ of a sub-production $\{A_{ij}\} \rightarrow NT_i$. This induction hypothesis is overlaid with the cross relations of $\mathcal{G}$ over the partition $\{A_{ij}, \{p\}\}$. The partition mapping algorithm then finds a mapping $\rho : \{p, A_{ij} \cap U_i^k\} \rightarrow \{T^k\}$.

Determining the induction hypothesis ν_{NT} for each breakdown non-terminal is similar to the way we constructed the non-terminal's constraining formula in Section 3.5. If a non-terminal does not come from the top non-terminal, a small number of extra terms $\{a_+\}$ are connected to the interface partition of NT. For example, in an unmarked list, a single marked element a can be added to the front to produce a marked list. The induction assumption ν_{NT}, then consists of ν, quantified over the interface partition $\{Q_1, Q_2\}$, with the added front element broken out. Recall that Q_1 is the singleton set of the second element of the augmented list, also the first element of the unmarked list. Q_2 is the rest of the unmarked list. Since every relation between a and Q_1 or Q_2 is known, all of the elements containing a are set to T or F depending on the relation with the interface partition. a does not appear in the final formula ν_{NT}.

ν, the induction hypothesis and the formula to prove, is already quantified over the action template partition $\{T\}$. Therefore, to break out any extra terms a from ν, Spec2Action has to make a guess about which partition element T_i a belongs to. It makes a guess and then attempt to do partition mapping. Often, partition mapping will fail. In this case, the grammar sub-productions themselves need to be extended. The grammar non-terminals can be extended indefinitely. Pivot terms are continually broken out of the sub-productions until a partition is produced that maps correctly to the goal partition and formula.

Thus, action template proving proceeds in a search tree. At each search node, a grammar production is broken out or a guess is made for breaking out a free variable into an action template partition element. The search tree is self-consistent if every possible grammar production has a partition mapping and every free variable breakout guess is justified by the partition mapping.

For removing from a linked list, there are eight final partition mappings in all. These are shown in Figures 4-7 and 4-8. Note that three of these examples are complete marked lists or unmarked sub-lists. Indeed, as the grammar breakouts proceed to deeper depths, more and more full, small examples are created. Thus, if

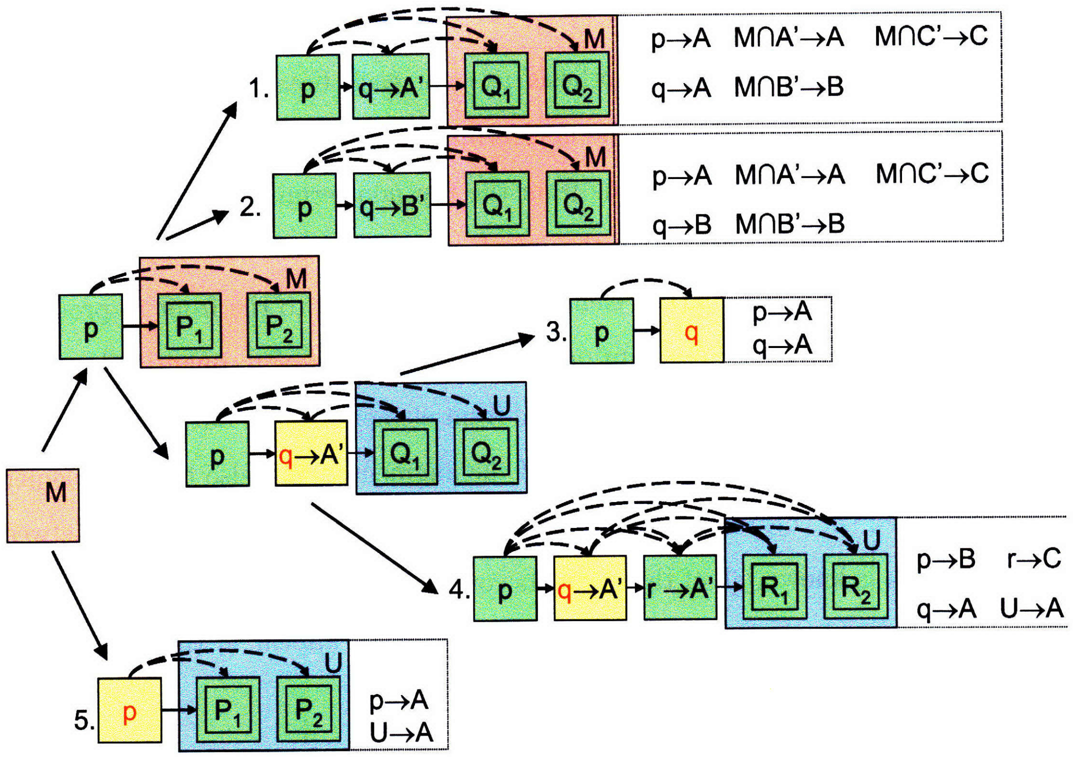

Figure 4-7: The five satisfying recursive partition mappings for the different ways a marked list breaks down and the ways the sub pivots could recursively map to $\{A', B', C'\}$.

there is a counter-example, it will eventually be discovered. In all of the example problems have considered, the sample set was always comprehensive enough that the hypothesized action template was correct. Theoretically, though, this counter-example could be fed back to the solve/generalize phase described in section 4.4.1, and a new action template could be hypothesized from the new knowledge.

4.4.5 Partition Mapping

The core of the recursive set builder is the partition mapper, a general algorithm that could be applied in other types of theorem proving and automated reasoning. Formally, it attempts to prove a particular goal formula G quantified over a goal partition $\{T^i\}$. The partition mapper is given a formula S, which is quantified over a different partition $\{U^j\}$. The mapper attempts to construct a mapping $\rho : \{U^i\} \rightarrow$

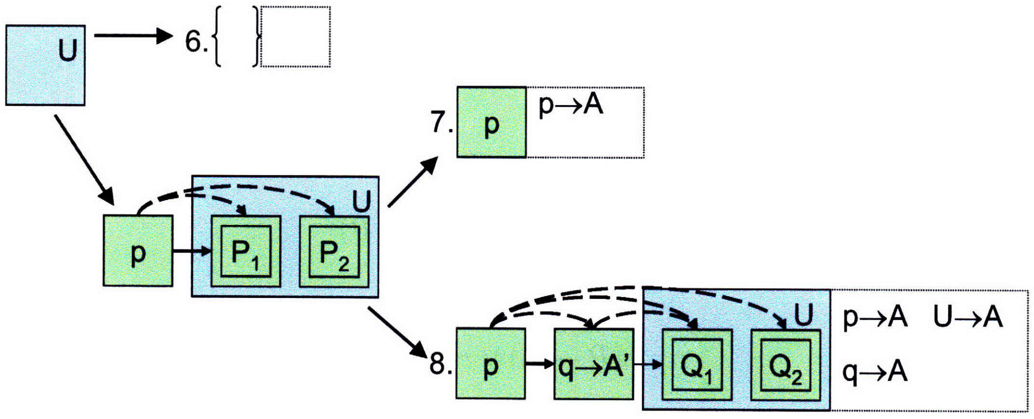

Figure 4-8: The three partition mappings for the ways an unmarked list breaks down and maps to $\{A, B, C\}$ from $\{A', B', C'\}$

$\{T^j\}$ such that $S \Rightarrow G : \{\rho^{-1}(T^j)\}$. As a simple example, consider the start and goal formulas

$$\begin{aligned} G &= \forall x : A\ \neg(marked\ x)\ \wedge\ \forall x : B\ (marked\ x) \\ S &= \forall x : P\ \neg(marked\ x)\ \wedge\ \forall x : Q\ \neg(marked\ x)\ \wedge\ \forall x : R\ (marked\ x) \end{aligned}$$

The correct ρ in this case is $\rho(P) = A; \rho(Q) = A; \rho(R) = B$. When G is quantified over $\rho^{-1} = \{P, Q, R\}$, the formula is

$\forall x : \{P, Q\}\ \neg(marked\ x)\ \wedge\ \forall x : R\ (marked\ x)$, which is the same as S. The partition mapping algorithm traverses the goal formula and produces a propositional formula defining all of the constraints that exist for mapping a set in S to a set in G. This formula is then solved with a SAT solver to get all of the possible satisfying mappings.

In the case of this example, the SAT formula for S to G would be:

$$\neg(R \rightarrow A) \wedge \neg(P \rightarrow B) \wedge \neg(Q \rightarrow B)$$

Furthermore, there is a SAT formula constraining that each set of S maps to exactly one set of G. The formula for this is:

$$\begin{array}{ll} & (R \to A) \vee (R \to B) \\ \wedge & \neg(R \to A) \vee \neg(R \to B) \\ \wedge & (P \to A) \vee (P \to B) \\ \wedge & \neg(P \to A) \vee \neg(P \to B) \\ \wedge & (Q \to A) \vee (Q \to B) \\ \wedge & \neg(Q \to A) \vee \neg(Q \to B) \end{array}$$

Together, these constraints force ρ to be the mapping above. The rest of this section will discuss the rules for producing the full constraining formula from $S : \{U^j\}$ to $G : \{T^i\}$. Let $\gamma(S, G)$ represent the partition map SAT formula. The first step is to normalize S into a disjunction of sub-formulas $S_1 \vee S_2 \vee \ldots \vee S_k$. Then, there is the constraint:

$$\gamma(S_1 \vee \ldots \vee S_k, G) = \gamma(S_1, G) \wedge \ldots \wedge \gamma(S_k, G)$$

There are two other simple boolean rules:

$$\begin{array}{rcl} \gamma(S, G_1 \wedge \ldots \wedge G_k) & = & \gamma(S, G_1) \wedge \ldots \wedge \gamma(S, G_k) \\ \gamma(S, G_1 \vee \ldots \vee G_k) & = & \gamma(S, G_1) \vee \ldots \vee \gamma(S, G_k) \end{array}$$

The interesting case is when G is a quantification, $\mathit{AtLeast}(n)\ x : T\ G(x)$ or $\mathit{AtLeast}(\mathit{AllBut}(n))\ x : T\ G(x)$, and $S = S_1 \wedge \ldots \wedge S_k$ is a conjunction of quantifications over $\{U^j\}$. First, consider $G = \mathit{AtLeast}(n)\ x : T\ G^o(x)$. In this case, the partition mapper looks at each combination from S of the form:

$$S_{i_1,\ldots,i_l} = \mathit{AtLeast}(n_1)\ x : U_1\ S^o_{i_1}(x) \wedge \ldots \wedge \mathit{AtLeast}(n_l)\ x : U_l\ S^o_{i_l}(x)$$

such that $n_1 + \ldots + n_l \geq n$. Usually, $n = 1$ and there is only one conjunction from S that is used. Now,

$$\gamma(S_{i_1,\ldots,i_l}, G) = (U_1 \rightarrow T) \wedge \ldots \wedge (U_l \rightarrow T) \wedge \gamma(S^o_{l_1} \wedge \ldots \wedge S^o_{l_i} \wedge UQ(U_1, \ldots, U_l), G^o)$$

If S has any universal quantifications over the sets U_j (i.e. of the form $\forall x : U_j\ S^o_j(x)$), these are also added to the conjunction, denoted by $UQ(U_1, \ldots, U_l)$. Also, if a set U_j is deduced to contain at least m elements, then a statement of the form $\mathit{AtLeast}(\mathit{AllBut}(n))\ x : U_j\ S^o_j(x)$ is converted to $\mathit{AtLeast}(m-n)\ x : U_j\ S^o_j(x)$, which is added to the set of available conjuncts of S.

$\gamma(S, G)$ is the disjunction of all of these possible combinations.

$$\gamma(S, G) = \bigvee_i \gamma(S_{i_1,\ldots,i_l}, G)$$

When $G = \mathit{AtLeast}(\mathit{AllBut}(n))\ x : T\ G(x)$, there is a complementary strategy. The partition mapper considers every combination from S of the form:

$$S_{i_1,\ldots,i_l} = \mathit{AtLeast}(\mathit{AllBut}(n_1))\ x : U_1\ S^o_{i_1}(x) \wedge \ldots \wedge \mathit{AtLeast}(\mathit{AllBut}(n_l))\ x : U_l\ S^o_{i_l}(x)$$

such that $n_1 + \ldots + n_l \leq n$. Unlike in the $\mathit{AtLeast}(n)$ case, each set U_j is tested and constrained. If U_j is not part of the combination, the partition mapper sets $S^o_{i_j} = \mathit{True}$.

Then, $\gamma_{U_j}(S_{i_1,\ldots,i_l}, G) = \neg(U_j \rightarrow T) \vee \gamma(S^o_{i_j} \wedge UQ(U_j), G^o)$. UQ is the conjunction of universal quantifications over U_j. The intuition is that for U_j to map to T, the constraints necessary to have $S^o_{i_j} \wedge UQ(U_j) \Rightarrow G^o : \{U\}$ must hold. Now, each set's constraints in the combination are conjoined together.

$$\gamma(S_{i_1,\ldots,i_l}, G) = \gamma_{U_1}(S_{i_1,\ldots,i_l}, G) \wedge \ldots \wedge \gamma_{U_l}(S_{i_1,\ldots,i_l}, G)$$

Finally, all possible combinations are disjoined to produce the final constraint formula

$$\gamma(S,G) = \bigvee_i \gamma(S_{i_1,\ldots,i_l}, G)$$

The usual case is $n = 0$, which is the universal quantifier $\forall$. In this case, there is always only one possible combination.

When G is a grounded expression at the root of the tree, $\gamma(S,G) = True$ if $S \Rightarrow G$, and $\gamma(S,G) = False$ otherwise. In this way, a boolean satisfiability formula is created describing all of the possible constraints on mapping from the sets of S, $\{U\}$, to the sets of G, $\{T\}$. As mentioned above, the propositional formula is constrained so that each U maps to exactly one T. If a satisfying mapping is found, then the partition mapper has demonstrated a way to construct the set partition $\{T\}$ such that the goal condition G holds over this partition. In this way, Spec2Action is able to prove the correctness of action templates by structural induction over the grammar of the start formula.

4.4.6 From partition maps to programs

When the partition mapper finds a satisfying assignment for its constraint formula, this assignment can be used to build recursive functions that build up the set partition $\{T\}$ of the action template. These recursive functions then become the parameters for the action template, and the result is a fully operational, correct action. This section examines how this works in the case of inserting an element into a list, and the case for removing an element from a list.

Consider inserting into a list. Spec2Action need to produce the two sets A and B to satisfy ν, Equation 4.4. B is the singleton set of the last element of the list. A is the rest of the list. The list has the grammar shown in Equation 4.3. There are two possibilities at the top level L_S and ϵ. If the list is ϵ, $A = \{\}$ and $B = \{\}$.

Otherwise, Spec2Action attempts to do partition mapping from $\{p, P_1 \cap A', P_1 \cap B', P_2 \cap A', P_2 \cap B'\}$ to $\{A, B\}$. The first attempt fails, so the non-empty production

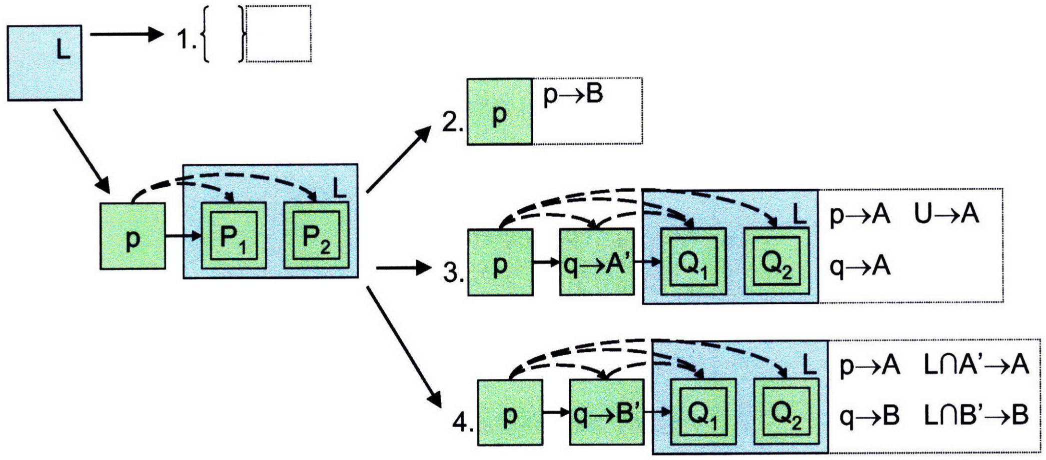

Figure 4-9: The four partition mappings for inserting an element at the end of a list

is broken down into the three possibilities shown in Figure 4-9. The first case (#2 in the figure) is the singleton list $\{p\}$. In this case, the partition mapping is $\{p\} \rightarrow B$. Thus, $A = \{\}$ and $B = \{p\}$. In the second case (#3 in the figure), Spec2Action has broken out a second free element q and two subsets $P_3$and P_4. This case also guesses that q maps to A in the induction hypothesis on $\{q, P_3, P_4\}$. The broken-out induction hypothesis is thus:

$$\begin{aligned} & \forall x : A' \ \exists y : \{A', B'\} \ (\mathit{connected}\ x\ y) \\ \wedge \ & \forall x : B' \ \forall y : \{A', B'\} \ \neg(\mathit{precedes}\ x\ y) \\ \wedge \ & \exists y : \{A', B'\} \end{aligned}$$

This hypothesis is overlaid with the grammar cross relations on $\{p, q, P_3, P_4\}$. (negatives omitted)

$$\begin{aligned} & (\mathit{precedes}\ p\ q) \wedge (\mathit{connected}\ p\ q) \\ \wedge\ & \forall x : P_3\ (\mathit{connected}\ p\ x) \\ \wedge\ & \forall x : P_4\ (\mathit{connected}\ p\ x) \\ \wedge\ & \forall x : P_3\ (\mathit{precedes}\ q\ x) \\ \wedge\ & \forall x : P_3\ (\mathit{connected}\ q\ x) \\ \wedge\ & \forall x : P_4\ (\mathit{connected}\ q\ x) \end{aligned}$$

In this case, the partition mapping succeeds. It is $\{p\} \rightarrow A$,$\{q\} \rightarrow A$,$A' \cap P_3 \rightarrow A$,$A' \cap P_4 \rightarrow A$, $B' \cap P_3 \rightarrow B$, $B' \cap P_4 \rightarrow B$. The third case (#4 in the figure) guesses that q maps to B. The broken-out induction hypothesis, overlaid on the breakdown $\{q, P_3, P_4\}$, is:

$$\begin{aligned} & \forall x : A'\ \exists y : \{A', B'\}\ (\mathit{connected}\ x\ y) \\ \wedge\ & \forall x : B'\ \forall y : \{A', B'\}\ \neg(\mathit{precedes}\ x\ y) \\ \wedge\ & (\mathit{None}\ B' \cap P_3) \\ \wedge\ & (\mathit{None}\ A' \cap P_3) \end{aligned}$$

The successful partition mapping for this case is $\{p\} \rightarrow A$,$\{q\} \rightarrow B$, $A' \cap P_4 \rightarrow A$, $B' \cap P_4 \rightarrow B$. This is a degenerate case where $A'$and $B'$are both empty and q is the last element of the list. The algorithm doesn't know this, but it does find a partition map that is logically correct. These four partition solutions now tell Spec2Action how to produce a recursive function s to generate A and B given a list ω.

The recursive function σ is as follows

```
$\sigma(\omega)$ = case $\omega$ of
        $\{\} \rightarrow A = \{\};$
                $B = \{\};$
        $\{p\} \rightarrow A = \{\};$
                $B = \{p\};$
        $\{p, q, P_3, P_4\} \rightarrow$
          if $q \in \sigma(\{q, P_3, P_4\}).A$ then
            $A = \{p, q, \sigma(\{q, P_3, P_4\}).A\};$
            $B = \{\sigma(\{q, P_3, P_4\}).B\};$
          else
            $A = \{p, \sigma(\{q, P_3, P_4\}).A \cap P_4\};$
            $B = \{q, \sigma(\{q, P_3, P_4\}).B \cap P_4\};$
```

Thus, partition mapping not only tells us that the partition to satisfy the action template $\{A, B\}$ exists, it contains the information necessary to build a recursive function to generate that partition. The case for removal from a linked list is more complex, but it follows the same pattern. The partition maps are shown in Figure 4-7. Recall that B is the singleton set of the element right in front of the marked element to be deleted, C is the singleton set of the element after the marked one, and A is the rest. The recursive function for $\{A, B, C\}$ is:

```
$\sigma(\omega)$ = **case** $\omega$ **of**
    $\{p;\ (marked\ p)\} \rightarrow A = \{p\}; B = \{\}; C = \{\}$
    $\{p, q;\ \neg(marked\ p),\ (marked\ q)\} \rightarrow$
      $A = \{p, q\};$
      $B = \{\};$
      $C = \{\};$
    $\{p, q, P_3, P_4;\ \neg(marked\ p),\ \neg(marked\ q)\} \rightarrow$
      **if** $q \in \sigma(\{q, P_3, P_4\}).A$ **then**
        $A = \{p, q, \sigma(\{q, P_3, P_4\}).A\};$
        $B = \sigma(\{q, P_3, P_4\}).B;$
        $C = \sigma(\{q, P_3, P_4\}).C;$
      **else if** $q \in \sigma(\{q, P_3, P_4\}).B$ **then**
        $A = \{p, \sigma(\{q, P_3, P_4\}).A\};$
        $B = \{q, \sigma(\{q, P_3, P_4\}).B;$
        $C = \sigma(\{q, P_3, P_4\}).C;$
    $\{p, q, r, P_3, P_4; \neg(marked\ p), (marked\ q), \neg(marked\ r)\} \rightarrow$
      $A = \{P_3, P_4, q\};$
      $B = \{p\};$
      $C = \{r\};$
    $\{p, q, P_3, P_4; (marked\ p), \neg(marked\ q)\} \rightarrow$
      $A = \{p, q, P_3, P_4\};$
      $B = \{\};$
      $C = \{\};$
```

4.5 Conclusion

This chapter has introduced a system, Spec2Action that performs a specific type of program synthesis. It takes a program specification in the form of a start formula S and a goal formula G, and, with no further information, produces an action to solve that program. The action is a special type of program that can represent the core of many larger programs. It consists of a partition of the world into roles and a list of set relations to change across those roles. The set relations are all of the relations to change to satisfy the goal formula G.

This chapter has shown the universality of actions by showing how they can be used to represent the core of a variety of simple data structure operations. In most cases, the action performs the entire operation.

Spec2Action generates examples and hypothesizes the simplest action template that solves all of the examples. It then tries to prove the action template correct by structural induction over the grammar for S produced by Form2Grammar, discussed in Chapter 3. Given the hypothesis action template, Spec2Action absorbs the changes into the start S. Then, it deep simplifies G given S. This defines the set of conditions that the action template set partition must satisfy. The goal of the prover at this point is to prove the existence of the partition of any world ωsatisfying S such that the goal conditions hold.

This partition is constructed by structural induction over the grammar of S. Spec2Action assumes the existence of the partition for the decompositions of the grammar, and that it satisfies some condition ν derived from the top condition and the non-terminal type of the decomposition. The inductive partition of the decomposition is then overlaid with the breakdown partition induced by the pivot decomposition rule itself. This overlay produces a class of building block sets, which can then be mapped into the partition of roles Spec2Action is trying to create. In this way, by induction, the action template is proven correct, and the partition can be constructed by recursive functions over the grammar.

The key algorithm in this structural induction is the partition mapper. It takes a given formula S quantified over one set partition $\{U\}$ and a goal formula G quantified over another set partition $\{T\}$. By traversing through G, the partition mapper produces a propositional formula constraining all of the possible transitions from U to T that would make G true given S. The resulting proposition is run through a SAT solver, and the satisfying assignment defines a set mapping $\rho : \{U\} \rightarrow \{T\}$ such that $S \Rightarrow G : \{\rho^{-1}(T)\}$.

The ultimate goal of Spec2Action is to perform a high level, abstract kind of program synthesis. Instead of worrying about the preconditions and side effects that programming with STRIPS actions entails, Spec2Action just tries to determine the relations of the world that need to change. It is assumed that the relations can be changed at any time independently of each other. Thus, Spec2Action, given a program specification, produces a template of *what* to change. It is then possible for another module to determine *how* to effect those changes using primitive actions. That is, Spec2Action is an important initial component in the full program synthesis problem. The next chapter will discuss HELPS, the final program synthesis system. While the initial version of HELPS does not directly use Spec2Action, the two systems are complementary. Spec2Action determines the relations to change to satisfy a specification, and HELPS sequences the primitive actions to make it happen.

Chapter 5

HELPS

5.1 Introduction

The final SSGP system is HELPS, which stands for Hallucinated-Example Led Procedure Search. It performs program synthesis of iterative programs using STRIPS actions as the primitive elements. STRIPS actions have pre-conditions, and they typically change more than one relation. Thus, the effects of one STRIPS action can be negated by subsequent actions down-stream. For this reason, the order of STRIPS actions within a plan or a program is very important. This contrasts with the circumstances of Spec2Action, which only manipulates one relation at a time.

However, STRIPS actions are a more realistic representation for the types of actions that an autonomous agent would have available to it. Like Form2Grammar and Spec2Action, HELPS uses examples to guide its search for a correct program, and it proves the programs' correctness with the same theorem provers described in the previous sections. A program is specified using a start formula S, a goal formula G, and a set of primitive actions. Example worlds ω are produced satisfying S. Then, HELPS uses a standard, grounded planner (SATPlan) to produce sample programs that use primitive actions to transform the world ω to a world ω' that satisfies G. HELPS reasons backwards from the goal formula G. Each iteration has a pre-condition to satisfy and a partial program. At each search iteration, it generates examples, solves the examples using ordinary planning, and generalizes a

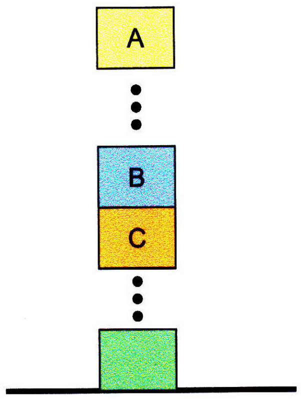

Figure 5-1: Unlike other general planning systems, there is no separate notion of *clear* or *atop*. $(clear\ A) = \forall x\, \neg(on\ x\ A)$. $(atop\ B\ C) = (on\ B\ C) \wedge \forall y\, \neg(on\ B\ y) \vee \neg(on\ y\ C)$.

program element to extend the current partial program. The search finishes when all examples satisfy the current pre-condition, and the current partial program becomes the hypothesis program. At this point, the theorem prover attempts to show that the pre-condition is implied by the starting formula. Otherwise, it generates a set of counter-examples, and the search begins again.

HELPS is capable of autonomously solving generalizations of some classic AI problems such as BlocksWorld and some other simple planning tasks. Because HELPS uses examples to guide its search, it has no need for specific domain knowledge. In the HELPS representation for BlocksWorld, there is one relation template $(on\ ?1\ ?2)$. That is, there are no special relations $(clear\ x)$ or $(atop\ x\ y)$ to denote x as the top block of a stack and to denote block x directly on top of block y, respectively (see Figure 1). Furthermore, there are no special BlocksWorld axioms or domain knowledge. All of the universe information comes from the specification of the problem and the specification of primitive actions.

Completed programs consist of primitive actions, while loops, and if-then conditions. The control formulas for the loop and the condition are standard quantified formulas.

Finding a correct program is a search over partial programs. From a particular partial program, the next step can be the invocation of a primitive action, the introduction of a condition, the focusing of the goal onto a sub-problem, the introduction of a fork, the closure of a fork, the breaking out of a specific term, the closure of a term break-out, the introduction of a loop, and the closing of a loop by finding a loop invariant. The next step to consider is determined by the format of the goal formula. When a particular partial program satisfies all of the examples, HELPS tries to show that the starting formula implies the precondition of the partial program. To do this, it uses the deep simplification and abstract induction theorem provers discussed in sections 3.3.7 and 3.3.8. Deep simplification reduces the problem by removing formulas that could be proven by ordinary first-order logic rules. Abstract induction proves the rest of the formula by using its counterexample search to prove facts that are only true in finite models and require a mathematical induction argument.

5.2 Representations

5.2.1 Primitive Actions

Before considering an example, let us examine how primitive actions and programs are represented. A primitive action is defined by its *preconditions* and its *outcomes.* The preconditions and oucomes are quantified formulas over the action's *parameters.* A parameter is either a single term or a set. An important restriction that HELPS requires is that the parameters must be distinct, and they must partition the world. This restriction significantly simplifies reasoning about actions. Also, it doesn't reduce the expressive power of the actions. Any STRIPS action can be broken up into one or more distinct HELPS actions that satisfy this constraint.

An action's precondition can be any quantified formula. However, the outcome quantified formula can only consist of simple relations, conjunctions, and universal quantifications. This enforces the constraint that actions be deterministic. Consider the actions $(moveTable\ a)$ and $(move\ a\ b)$, which moves block a onto the table or onto block b respectively. $moveTable$ is parameterized by the single object a and two sets A and B. A is the set of blocks under a, and B is all the other blocks in the world. The precondition for *moveTable* is:

$$\begin{aligned} & \forall x\ \neg(on\ x\ a) \\ \wedge\ & \forall x : A\ (on\ a\ x) \\ \wedge\ & \forall x : B\ \neg(on\ a\ x) \end{aligned} \tag{5.1}$$

The outcome of *moveTable* is

$$\forall x : A\ \neg(on\ a\ x) \tag{5.2}$$

That is, a is no longer on top of every element of A. It is no longer on anything, having moved to the table. $(move\ a\ b)$ is parameterized by two objects a and b and three sets A, B, and C. A is the set of blocks under a. B is the set of blocks under b. C is everything else. The precondition is:

$$\begin{aligned} & \neg(on\ a\ b) \\ \wedge\ & \forall x\ \neg(on\ x\ a) \\ \wedge\ & \forall x\ \neg(on\ x\ b) \\ \wedge\ & \forall x : A\ (on\ a\ x) \\ \wedge\ & \forall x : B\ (on\ b\ x) \\ \wedge\ & \forall x : C\ \neg(on\ a\ x) \wedge \neg(on\ b\ x) \end{aligned}$$

The outcome is

$$\begin{aligned} & (on\ a\ b) \\ \wedge\ & \forall x : A\ \neg(on\ a\ x) \\ \wedge\ & \forall x : B\ (on\ a\ x) \end{aligned}$$

a is removed from atop the tower of blocks A and moved onto the tower of blocks B.

5.2.2 Programs

This section shows how primitive actions can be combined to make full programs. There are five program elements: the primitive action call, the if-then statement, the while loop, the let statement, and the sequence. The primitive action call instantiates the parameter terms of an action with specific arguments. An argument is specified as a quantified formula $\phi(x)$ over a free term x. This means that any term a that satisfies $\phi(a)$ could be applied as the argument to the action's parameter term. For example, as part of clearing b, we want to move the top block on b to the table. This is specified as $(\mathit{moveTable}\ a) :\ (\mathit{on}\ a\ x) \wedge \forall y\ \neg(\mathit{on}\ y\ a)$. The set parameters A and B are defined by the precondition in terms of the argument a. When a hypothetical program executor encounters an action call, it would search the current state for an object satisfying the argument condition, and then it would execute the action on that term.

Action calls are combined via sequences, if-then-else statements, let statements, and while loops. A sequence consists of two programs executed one after the other. An if-then-else statement has two body programs and a condition quantified formula. An executor would evaluate the condition on the current world. If the condition was true, it would execute the first body program. Otherwise, if an else body exists, it would execute that instead. If there is no body, it does nothing. The next program component is a while loop. It consists of a quantified formula as a while condition and a sub-program as a body. The executor would evaluate the condition formula. If the condition held true, it would execute the body program. Then, it would evaluate the loop condition again. The body gets executed until the loop condition becomes false. Finally, there is a **let** statement, consisting of a parameter/argument list and a body. It simply creates an environment defining parameter terms and parameter sets which can be used in the body program. These statements are summarized in the grammar below:

$$\begin{aligned}
\mathit{Prog} &::= \mathit{Call} \mid \mathit{IfThen} \mid \mathit{Loop} \mid \mathit{Seq} \mid \mathit{Let} \\
\mathit{Loop} &::= \mathbf{while}(QF)\ \mathit{Prog} \\
\mathit{IfThen} &::= \mathbf{if}\ (QF)\ \mathbf{then}\ \mathit{Prog}\ [\mathbf{else}\ \mathit{Prog}] \\
\mathit{Seq} &::= \mathit{Prog}\ ;\ \mathit{Prog} \\
\mathit{Call} &::= \mathit{Action}(\mathit{Arg}, \ldots, \mathit{Arg}) \\
\mathit{Let} &::= \mathbf{let}\ P_1 = \mathit{Arg}_1, \ldots, P_k = \mathit{Arg}_k\ \mathbf{in}\ \mathit{Prog} \\
\mathit{Arg} &::= QF(x)
\end{aligned}$$

The specification for a program consists of a start formula S, a goal formula G, and a set of action definitions. In all of the BlocksWorld examples, there is a common start formula, constraining the relations among the blocks. This formula is:

$$\begin{aligned}
& \forall x \forall y\ (\mathit{on}\ x\ y) \Rightarrow \neg(\mathit{on}\ y\ x) \\
\wedge\ & \forall x \forall y \forall z\ ((\mathit{on}\ x\ y) \wedge (\mathit{on}\ y\ z)) \Rightarrow (\mathit{on}\ x\ z) \\
\wedge\ & \forall x \forall y \forall z\ ((\mathit{on}\ x\ z) \wedge (\mathit{on}\ y\ z)) \Rightarrow (\mathit{on}\ x\ y) \vee (\mathit{on}\ y\ x) \\
\wedge\ & \forall x \forall y \forall z\ ((\mathit{on}\ z\ x) \wedge (\mathit{on}\ z\ y)) \Rightarrow (\mathit{on}\ x\ y) \vee (\mathit{on}\ y\ x)
\end{aligned} \tag{5.3}$$

None of these formulas are redundant. They ensure that stacks of blocks cannot form loops, and that for every pair of blocks x and y in a stack, either $(\mathit{on}\ x\ y)$ or $(\mathit{on}\ y\ x)$. The goals for the three BlocksWorld problems of clearing block b, moving block a onto b, and moving a directly onto b are, respectively:

$$\begin{aligned}
\textit{Clear Block Goal} &= \forall x\ \neg(\mathit{on}\ x\ b) \\
\textit{Move Onto Goal} &= (\mathit{on}\ a\ b) \\
\textit{Move Directly Onto Goal} &= (\mathit{on}\ a\ b) \land (\forall x\ \neg(\mathit{on}\ a\ x) \lor \neg(\mathit{on}\ x\ b)) \\
\textit{Put All Blocks in a Tower} &= \forall x \forall y\ \neg(\mathit{on}\ x\ y) \lor \neg(\mathit{on}\ y\ x)
\end{aligned} \tag{5.4}$$

5.3 Challenge problems

To test the system and guide development, the exercise problems are the BlocksWorld applications above and a few other domains. They are general problems with simple specifications from a wide range of situations. The variety of domains where this type of program synthesis applies illustrates the power of the quantified formula representations and reasoners. The following are the programs that HELPS has sought to synthesize. They are illustrated in Figures 5-2 to 5-9.

1. Clearing a block b
2. Moving block a onto block b
3. Moving block a directly onto block b
4. Arranging all blocks into a single tower
5. Setting a row of switches: There are four primitive actions. Left and right actions to move a cursor up and down the row. An up and down action to set the current switch up or down respectively.
6. Moving across a two-dimensional grid. The primitive actions are moving up, down, left, and right.
7. Picking up an object and moving it somewhere else on a one-dimensional line. Primitive actions are left, right, pick-up, and drop.

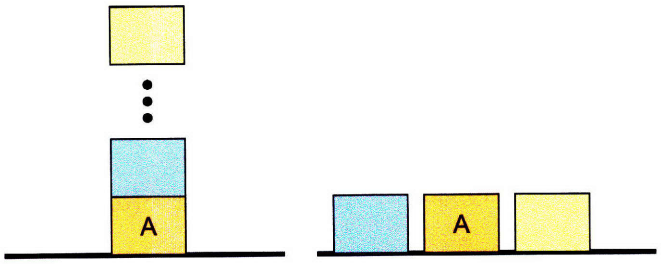

Figure 5-2: Problem: clearing block A

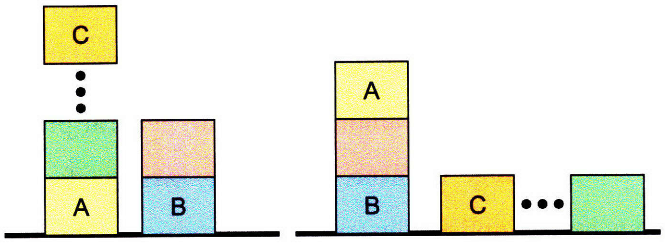

Figure 5-3: Problem: moving block A onto block B

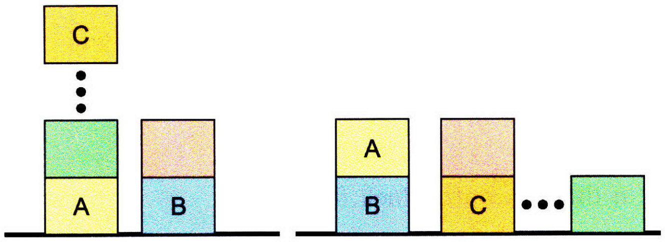

Figure 5-4: Problem: moving A directly onto B

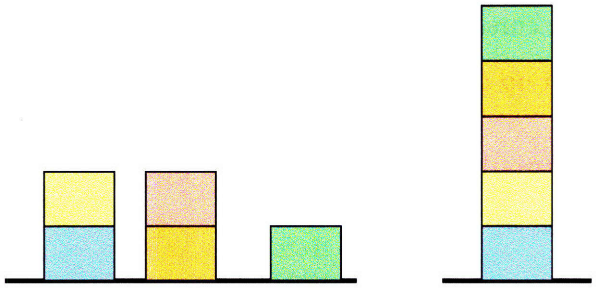

Figure 5-5: Problem: stacking blocks in a table in one stack

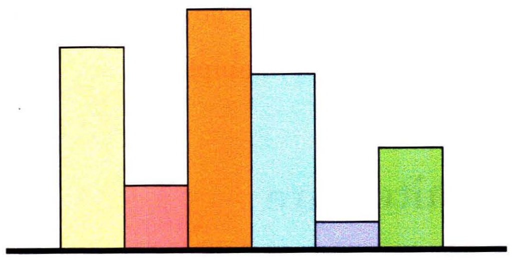

Figure 5-6: Problem: sorting a list of elements

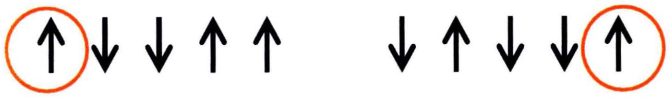

Figure 5-7: Problem: setting a row of switches

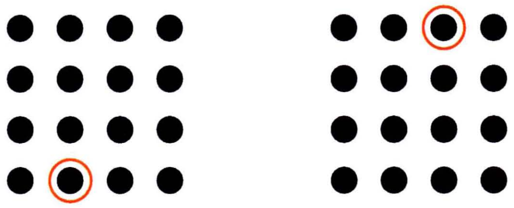

Figure 5-8: Problem: navigating a grid

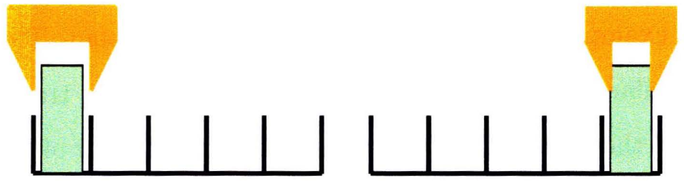

Figure 5-9: Problem: picking up an object and moving it

All of these problems and their solutions can be represented naturally with HELPS programs. The following section will show how programs can be synthesized by a search over partial programs, guided by examples.

5.4 HELPS Architecture

Fundamentally, HELPS is an iteration over partial programs. HELPS starts with an empty partial program with the specified start and goal formulas S and G. Through partial program transformations, HELPS moves in the direction of a successful program. This section describes the information in partial programs, and the transformations made on them.

5.4.1 Partial programs

Partial programs are incomplete programs ψ that are grown from the goal backwards to the start condition. Every partial program has an open goal to be satisfied. During the search, a partial program has its goal transformed and structural program elements are added. The initial partial program is an empty program with the goal specification G as the open goal. The search finishes when it finds a partial program whose open goal is implied by the initial start condition.

To understand the promise and complexity of an individual partial program, examples are used. A set of initial examples is generated using the start formula. The example generation algorithm is the same as the one used in Section 3.3.2. Deeper in the search, a partial program can acquire its examples in different ways depending on the transformation. For each new partial program, HELPS uses ordinary STRIPS planning on each example to produce the smallest example plan. The assumption is that something similar to this example plan would be the output trace of the program needed to complete the present partial program. The complexity of a partial program is determined by the average length of its example plans.

To do the grounded planning, HELPS uses its own version of SATPlan. That is, for a given example ω, a goal formula G, and an integer n, a propositional formula is produced constraining all of the possible plans of size at most n. This formula is solved with a SAT Solver, and the solution contains the correct plan.

Some of the SATPlan constraints are that at most one action can be executed at a time. To be executed, an action's preconditions must be satisfied. If an action is executed, all of its outcome relations must change, and all of the other relations must stay the same.

Using binary search on n, the plan length, a minimal plan is found, and this becomes the guiding example plan.

Very often, the open goal for a partial program will be a conjunction of formulas $\phi_1 \wedge \phi_2$. In this case, it is difficult to know which formulas to focus on. Do we solve ϕ_1 first or ϕ_2? The solution to one formula may conflict with the solution to the other, so the two problems cannot be solved independently. This is known as the Sussman anomaly[38]. However, HELPS cannot accurately guess a priori whether one of the formulas should be the last one solved. A formula ϕ is called a *final sub-goal* if it is satisfied on the last step of a plan, but it is not satisfied on the pen-ultimate step, time $n-1$. For any minimal plan with $n > 0$ satisfying $\phi_1 \wedge \phi_2$, either ϕ_1 or ϕ_2 must be a final sub-goal.

So, in addition to an open goal G, a partial program may also be constrained by a final sub-goal FG. Each of the example plans must either be of length 0, or they must not satisfy FG at the pen-ultimate step while satisfying it at the last step. The SATPlanner is easily modified to constrain a plan to have both a goal and a final sub-goal. Final sub-goals are an important concept. During the partial program search, if a partial program has a final sub-goal in its open goal, then the search chooses the partial program transformation based on the final sub-goal, not the goal itself. However, the partial program transformation, when it runs, must operate on and solve the whole goal, not just the final sub-goal.

In other words, the final sub-goal guides the solution search, but the total goal is always the result that is sought.

To summarize, every partial program ψ has an open goal G, a set of starting examples Ω, and the incomplete program itself μ. The partial program may have a

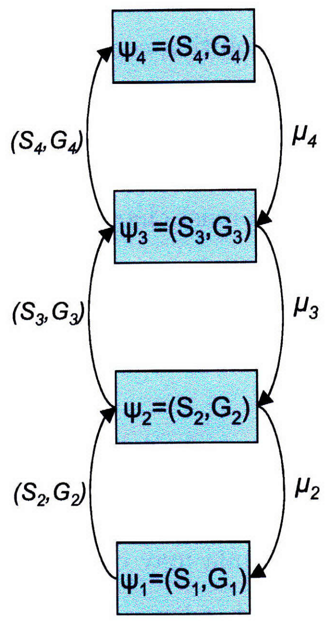

Figure 5-10: Partial programs consist of a chain of transformed problems. The output program is constructed from the top down.

starting formula which generated the examples, or the examples may be provided, as they are done in the production of the bodies of loops. The partial program may also have a final sub-goal.

5.4.2 Transforming partial programs

Program synthesis search proceeds by transforming partial programs ψ, ultimately searching for a partial program whose open goal is implied by the initial starting condition. There are nine possible transformations: introducing an action call, introducing a final sub-goal, introducing a condition, introducing a fork, closing a fork, introducing a free term, closing a free term, introducing a loop, and closing a loop. These will now be examined in detail. A partial program maintains a link to the previous partial program it was derived from. From this chain of links, HELPS can derive a complete program, because each link in the chain knows how to produce a program out of the program produced by its child. See Figure 5-10.

Introducing an action call

The production of action calls is provided by a module called the call mapper. An action call is attempted if the current open goal is a single relation, or if the current final sub-goal is a single relation. The call mapper attempts to match the goal $G(\psi)$ of the existing partial program ψ with the outcome of one of the primitive actions. The precondition for the action then becomes the open goal of the new partial program $G'(\psi')$.

Recall that the outcome of an action consists entirely of conjunctions and universal quantifications. Therefore, its changes can be represented as a list of set relations over the action's parameter partition. Set relations are used to represent programs in Spec2Action, where an action template consists of a partition and a set of change set relations over this partition. The representation is discussed in Section 4.3. Basically, a set relation represents the set of all relations between objects of different roles. For example, in the *moveTable* action of BlocksWorld, the set relation outcome is $\neg(on\ a\ A)$, where A is defined as the set of blocks that a is on before the *moveTable* is invoked.

Using the set relations, HELPS analyzes the goal tree to determine a pre-action formula G'. When the action (*moveTable a*) is applied to a world ω'satisfying G', then the subsequent state ω must satisfy G. Finding G' is done by a special algorithm called the Change Manager, which is discussed in the next section. The Change Manager is able to propagate formulas forward and backward through actions. In this case, G is propagated backward through the *moveTable* action to obtain a formula G' that would produce G.

Because actions are defined over their parameter partition, the change-propagated goal formula will contain action parameter terms and parameter sets. These must be eliminated for the new pre-condition to make sense. Fortunately, parameters of an action call can be eliminated from a formula in a straightforward and deterministic way.

For example, consider the BlocksWorld goal formula $G = \forall x\ \neg(on\ x\ b)$. Propagated backward through *moveTable*, this becomes:

$$\begin{aligned} G' = \quad & \forall x : \{A, B\}\ \neg(on\ x\ b) \\ \wedge \quad & (A\ b)\ \vee\ \forall x : \{a\}\ \neg(on\ x\ b) \end{aligned}$$

$\{a\}$ is the singleton set of the parameter term a. That is, every object besides the parameter object a must not be on b. The parameter a must either not be on b, or b must be a member of A, i.e. under a. This is a tautology, but the Change Manager doesn't necessarily know it. Intruitively, for *moveTable* to solve this goal G, either all blocks are already not on b, or there is exactly one block on b.

Eliminating the parameter sets is easy, because they are defined in the action precondition. In the BlocksWorld example, $A = x : (on\ a\ x)$. $B = x : \neg(on\ a\ x)$. When these are eliminated, the formula becomes

$$\begin{aligned} G' = \quad & \forall x\ \neg(on\ x\ b) \\ \wedge \quad & (on\ a\ b)\ \vee\ \forall x : \{a\}\ \neg(on\ x\ b) \end{aligned}$$

Now, eliminating the parameter term a depends on whether $a = b$ or $a \neq b$ in the call. Intuitively, $a = b$ is a poor choice, since b is the block we are trying to clear in the first place. a should be some block on top of b. However, HELPS doesn't know this a priori, so it must consider both cases. When $a = b$, every occurrence of a can be replaced with b. In this case, $\forall x : \{a\}\ \neg(on\ x\ b)$ is trivially true, since its meaning is for all x *not equal to* b in the set $\{a\}$, $\neg(on\ x\ b)$. Since there are no such x, the $\forall$ quantification is trivially true. $G' = \forall x\ \neg(on\ x\ b) \wedge (F \vee T) = \forall x\ \neg(on\ x\ b)$. In the case where $a \neq b$, $G' = \forall x\ \neg(on\ x\ b) \wedge ((on\ a\ b) \vee \neg(on\ a\ b)) = \forall x\ \neg(on\ x\ b)$. This statement means for all x *not equal to* a, $\neg(on\ x\ b)$.

The action-transformed goal conditions must now be combined with the action's precondition, Equation 5.1. With the set conditions removed, this precondition is

$\forall x\ \neg(on\ x\ a)$. The precondition must incorporate the goal free variable b. If $a = b$, the precondition is $\forall x\ \neg(on\ x\ b)$. If $a \neq b$, the precondition with b broken out is $\forall x\ \neg(on\ x\ a) \wedge \neg(on\ b\ a)$. At this point, there are two new open goals, depending on whether $a = b$ or not. They are $G_1' = \forall x\ \neg(on\ x\ b)$ and $G_2' = \forall x\ \neg(on\ x\ a) \wedge \neg(on\ b\ a) \wedge \forall x \neg(on\ x\ b)$. Any $a \neq b$ that satisfies this second condition can be used to make the goal true. Hence, to finally eliminate a, we add an existential quantifier to the front of G_2',

$$\begin{array}{lcl} \exists a & & \forall x\ \neg(on\ x\ a) \\ & \wedge & \neg(on\ b\ a) \\ & \wedge & \forall x \neg(on\ x\ b) \end{array}$$

Thus, there are now two action calls that can be used. One where $a = b$, and the other one where a is the term such that $G_2'(a)$ holds. If both of these new goals seemed like plausible paths, then we would create an if-then-else statement with two calls. It would be:

```
if $\forall x\ \neg(on\ x\ b)$ then
  $(moveTable\ b)$
else
  $(moveTable\ a) : (G_2'\ a)$
```

The new open goal would be $G_1' \vee \exists a\ G_2'(a)$. However, since $\forall x\ \neg(on\ x\ b)$ is the same as the original goal condition, HELPS automatically prunes this path and returns the more interesting single $(moveTable\ a)$ call with the open condition $\exists a\ G_2'(a)$. Intuitively, this search direction means that to clear a block b, one must clear b of all but one block.

As this example shows, dealing with free variables in the goal in combination with an action's parameters can be tricky. Mapping action postconditions is one of the cases where equality between terms must be considered. However, the different equality mappings can be dealt with in a straightforward and consistent way. It shows the robustness of the quantified formula representation, and the simplicity of assuming that all terms are not equal to each other by default.

When considering a primitive action, the call mapper always ensures that the action does something to reduce the goal. If the goal is passed through to become an open goal of the new partial program, the action was considered to be ineffective, and the new partial program is thrown out. Furthermore, the action must accomplish something to reduce the final sub-goal if the current partial program has one. In this way, the final sub-goal of a partial program serves to focus the attention of the program search only on those actions that could make a difference to that sub-goal.

The transformed program has the old starting conditions and the new open goal. The existing partial program produces its solution program from the solution of the transformed program by appending the action call in a sequence. That is, $\mu(\psi) = \mu'(\psi');(\textit{Action Call})$.

To summarize, the call mapper takes the goal and final sub-goal of an existing partial program, and it tries to apply a primitive action to reduce this goal in some way. The reduced goal, in combination with the precondition of the action then become the next partial programs that are added to the search queue.

Introducing an if-then condition

For solving any partial program ψ, one can always assume that the open goal $G(\psi)$ is false at the beginning of execution. If the open goal were true, there is no need to execute any actions, and execution can stop immediately. Thus, a simple partial program transformation is to introduce the negation of the goal as an additional starting condition. The solution from the new partial program is then wrapped with an if-then statement with $\neg G$ as the if-then condition.

For example, consider the goal $G = \forall x\ \neg(\mathit{on}\ x\ b)$ with the start formula S. If $S \not\Rightarrow \forall x\ \neg(\mathit{on}\ x\ b)$, then the transformed start condition is $S' = \exists x\ (\mathit{on}\ x\ b) \wedge S$. The new partial program has start formula $S'$and goal G. If the current partial program has no start formula, only starting examples ω, each example ω is tested on G. If some of the examples satisfy G, a new partial program ψ' is created consisting only of those examples not satisfying G.

The existing partial program produces its output solution from the new partial program solution by wrapping an if statement around it. $\mu(\psi) =$ **if** $\neg G$ **then** $\mu'(\psi')$.

Introducing a final sub-goal

As mentioned previously, a final sub-goal serves to focus the attention of the program search. When a partial program ψ has a final sub-goal FG, any action hypothesized for the end of $\mu(\psi)$ must make some progress against FG. So, if a partial program ψ has a goal $G = G_1 \wedge \ldots \wedge G_k$ and no final sub-goals, then each example is tested to see which goals G_i can be final sub-goals in their minimum plans. If there is one conjunct G_i that is a final sub-goal for all examples, then this becomes the next partial program, denoted $G_1 \wedge \ldots \wedge G_k; G_i$.

On the other hand, if different samples have different final sub-goals, then a branch or fork is introduced. For example, imagine if samples $\{\omega_{k_i}\}$ had final sub-goal G_i and samples $\{\omega_{m_j}\}$ had final sub-goal G_j. In this case, two sub-problems are created, ψ'_1 and ψ'_2. $G_1 \wedge \ldots \wedge G_k; G_i$ with sub-goal G_i is the goal for ψ'_1 with samples $\{\omega_{k_i}\}$ as the given. $G_1 \wedge \ldots \wedge G_k; G_j$ with sub-goal G_j is the goal for ψ'_2 with samples $\{\omega_{k_j}\}$as the given. Note that introducing a fork produces two sub-problems with only samples and no starting formulas. HELPS produces solutions to the two sub-problems that solve the examples, but the solutions cannot be proven until the fork is closed back up again.

Let μ'_1 be the solution to ψ'_1 with pre-condition θ'_1 (that is, all the examples in ψ'_1 satisfy θ'_1 and are solved by μ'_1) and μ'_2 be the solution to ψ'_2 with pre-condition θ'_2. The resulting program is

```
μ = if $\theta'_1$ then
        $\mu'_1$
    else
        if $\theta'_2$ then
            $\mu'_2$
```

The pre-condition for ψ is $\theta = \theta'_1 \vee \theta'_2$. The output for more than two cases is analagous.

Introducing and closing a branch

Branches can also be introduced if the goal or final sub-goal of a partial program is a disjunction of the form $G_1 \vee \ldots \vee G_n$ or $G; (G_1 \vee \ldots \vee G_n)$ respectively. In this case, each sample is tested to see which goals G_i can be satisfied in the minimal sized plan. If there is one goal G_i that all samples can satisfy, then the next partial program is G_i or $G \wedge G_i$ respectively.

However, if different samples satisfy different G_i's in their minimal plans, then a branch is introduced. The process is exactly analogous to the case when different samples seek different final sub-goals when the goal is a conjunction. For example, the sample set may be split into two groups $\{\omega_{k_i}\}$ and $\{\omega_{k_j}\}$ for two disjuncts G_i and G_j which are part of the final sub-goal for the larger problem $G; G_1 \vee \ldots \vee G_n$. These are put into two partial program problems $\psi'_1 = (\{\omega_{k_i}\}, G \wedge G_i)$ and $\psi'_2 = (\{\omega_{k_j}\}, G \wedge G_j)$. If these two problems are solved with programs and pre-conditions μ'_1,μ'_2,θ'_1,θ'_2 respectively, then the program solution μ for ψ is, just like with the branched final sub-goal:

```
μ  =  if $\theta_1'$ then
          $\mu_1'$
      else
          if $\theta_2'$ then
              $\mu_2'$
```

The pre-condition is $\theta = \theta_1' \vee \theta_2'$.

Introducing and closing out a free term

Free terms are broken out whenever the goal or the final sub-goal is of the form $AtLeast(n)\ x\ \phi(x)$. An example such formula is $\exists x\ \phi(x)$. In this case, each object t_{ij} of each sample ω_i is considered. If $\phi(t)$ can be solved by the minimum plan length, then t is a candidate for being broken out. A candidate object t_i is produced for each sample ω_i. Then, a fresh term f is created and substituted into each sample, producing $\omega_i' = \omega_i[t_i|f]$. These samples lead to the next partial program $\psi' = (\{\omega_i'\}, \phi(f))$. If $\exists x\ \phi(x)$ is only a final sub-goal, where the goal is $G = O \wedge \exists x\ \phi(x)$, then f must be broken out of O, a process described in Section 2.3.

Let μ' and θ' be the program solutions and pre-conditions respectively for ψ'. Then, the solution for ψ is:

```
μ  =  let $f = f.\theta'(f)$
          $\mu'$
```

The pre-condition $\theta = \exists f\ \theta'(f)$

Introducing a loop

Loops are the most complex structural element to construct and reason about. A loop is created to solve a goal or a final sub-goal with an *AllBut* quantification, usually $\forall$. A sub-problem is created to fix each term not satisfying the quantification one by one. The solution to this sub-problem becomes the body of the produced loop. More preceisely, let the final sub-goal $FG(\psi) = \mathit{AllBut}(m)\ x\ \phi(x)$ and $G(\psi) = FG(\psi) \wedge OG(\psi)$. If there is no final sub-goal, $OG(\psi) = T$. HELPS creates an environment with a new set partition of two sets A and B. $A = x : \phi(x)$. $B = x : \neg\phi(x)$. The new partial program has the goal $G'(\psi') = \forall x : A\ \phi(x) \wedge \exists x : B\ \phi(x)$. That is, the sub-program must find one term x in B for which it can reverse the value of $\phi(x)$. At the same time, the sub-program must not back-slide, so it must maintain that every term in A continues to satisfy $\phi(x)$. There are additional constraints to make sure that $OG(\psi)$ is true after the program as well.

Like branches and term introductions, loop body programs have no start formulas, only examples which an output program must solve. The example problems ω' for ψ'come from the example problems ω_j of ψ. Each example plan is examined. Consider a single plan p_j with action sequence $\alpha_{j,1}, \dots, \alpha_{j,n}$ and intermediate state sequence $\omega_{j,0}, \dots, \omega_{j,n}$, $\omega = \omega_0$. At time 0, there are a certain number of terms $x_1, \dots, x_k$ which satisfy $\neg\phi(x_i)$. At time n, at most m terms do. The example plan is analyzed to find those times t_i where a particular term x_i changes from $\neg\phi(x_i)$ to $\phi(x_i)$ and continues to satisfy $\phi(x_i)$ at all future times where a further $x_{i+1}, \dots x_{k-m}$ change from $\neg\phi(x)$ to $\phi(x)$.

In this way, a list of example plans is created $\alpha_{j,t_{i-1}+1}, \dots, \alpha_{j,t_i}$ with the example sequences $\omega_{j,t_{i-1}}, \dots, \omega_{j,t_i}$. So, the example set for ψ'consists of $\{\omega_{j,t_{i-1}}\}$ with the goal $G'(\psi') = \forall x : A\ \phi(x) \wedge \exists x : B\ \phi(x) \wedge (OG(\psi) \vee \exists x : B\ \neg\phi(x))$. The examples are augmented with the set partition $\{A, B\}$. That is, every every term x in every example $\omega_{j,t_{i-1}}$ is marked $(A\ x)$ or $(B\ x)$ depending on whether $\phi(x)$ or $\neg\phi(x)$. From the perspective of ψ',A and B are considered to be ordinary sets without any specific meaning. The goal is defined in terms of A amd B, all of the examples are labeled A

and B, and that is all that ψ' knows.

If the partial program produces an output program $\mu'(\psi')$, it is incorporated into $\mu(\psi)$ in the following way:

$$\begin{aligned}\mu(\psi) \;=\; & \textbf{while}\ \neg G(\psi) \\ & \quad \textbf{let}\ A = x : \phi(x),\ B = x : \neg\phi(x)\ \textbf{in} \\ & \quad\quad \mu'(\psi')\end{aligned}$$

Since ψ' does not have a starting formula, only initial examples, it has no way of knowing if it has found a correct program. All it can know is if its program solves all of the examples or not. If its program does solve all examples, it potentially has an effective loop body to be used in the wider loop, which in turn hopefully solves a major part of the original goal of ψ, $G(\psi)$.

Jumping out of a loop

When a loop body is hypothesized as finished, the outer partial program must determine what the loop, as a whole, accomplishes. Let the outer partial program be ψ and let the loop body partial program be ψ'. The completed ψ' has an open precondition G^o. This open precondition is satisfied by the examples of ψ', but more analysis is needed to determine a new precondition open goal $\bar{G}$ such that if ψcan show a program $\bar{\mu}$ that produces $\bar{G}$, then $\bar{\mu}$; **while** $\neg G(\psi)$ $\mu'(\psi')$ is a program that produces $G(\psi)$.

Specifically, G^o cannot necessarily serve as the precondition of the loop. Since $G^o$is the open goal for the loop body program, it must hold for every iteration of the loop. So, it is not enough to ensure that G^o holds before the first iteration. It must hold every time the loop comes around. This is illustrated in Figure 5-11. The problem is to find $\bar{G}$ such that $G^o \vee G(\psi)$ holds at each iteration. This is the problem of finding a loop invariant. In general, this is a difficult logical problem without a simple, algorithmic solution. HELPS makes various assumptions and approximations

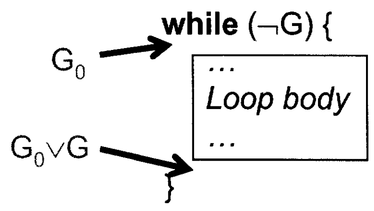

Figure 5-11: Loop invariants must hold at the beginning and end of every loop.

to make the invariant finding possible.

A critical module in finding the loop invariant is the change manager. This routine can propagate a quantified formula ϕ forward and backward through a program μ. We denote $\tau(\mu, \phi)$ to be the forward formula and $\tau^{-1}(\mu, \phi)$ to be the backward formula. The forward propagation means that for any world ω, if $\phi(\omega)$ holds at a particular time, and μ is applied to ω, then $\tau(\mu, \phi)$ holds on $\mu(\omega)$. Backward propagation means that for any world ω satisfying $\tau^{-1}(\mu, \phi)$, $\mu(\omega)$ satisfies ϕ. $\tau^{-1}(\mu, \phi)$ is the formula that must hold on ω for ϕ to hold when $\mu$is applied to ω.

Now, the problem of finding a loop invariant can be stated as: find I such that $I \Rightarrow \tau^{-1}(\mu', I) \wedge (G^o \vee G(\psi))$. Such an I either satisfies G^o or $G(\psi)$ at the beginning of the loop. Furthermore, since it implies $\tau^{-1}(\mu', I)$, we know that I holds after μ' is run. Therefore, I satisfies G^o or $G(\psi)$ at every iteration of the loop. Thus, I is our candidate for $\bar{G}$. the goal of the loop invariant finder is to find the minimally sufficient I satisfying the above implication.

The change manager module and the loop invariant finder are discussed in the following section. They produce the open goal $\bar{G}$ of the pre-loop partial program $\bar{\psi}$. $\bar{G}$ satisfies $\bar{G} \Rightarrow \tau^{-1}(\mu', \bar{G}) \wedge (G^o \vee G(\psi))$. The starting conditions (examples and potential formula) for $\bar{\psi}$ are the same as the starting conditions for ψ. If $\bar{\psi}$ produces a solution program $\bar{\mu}$, then the solution program for ψ is:

```
$\mu(\psi) \; = \; \bar{\mu}(\bar{\psi})$;
                 while $\neg G(\psi)$
                   let $A = x.\phi(x),\ B = x.\neg\phi(x)$ in
                     $\mu'(\psi')$
```

Finishing the program

At some point in the program search, the initial partial program will have all of its initial examples solved. The current partial program ψ will have the initial start formula S and an open goal $\bar{G}$, where every initial example satisfies $\bar{G}$. At this point, HELPS uses theorem proving to try to show that $S \Rightarrow \bar{G}$. It first deep simplifies $\bar{G}$ given S. Then, it attempts to prove the deep simplified $\bar{G}$ by an abstract induction counterexample search, described in Section 3.3.8. If the proof succeeds, then the program search is complete. The solution for ψ is the empty program ; . This solution is propagated downward through the chain of partial programs that led to ψ. The program produced at the end of this chain μ is the output of HELPS, and this program provably satisfies its spec. An example will now be shown to demonstrate how the search elements work together.

5.4.3 Example: Clearing a Block

This example will show the program search for one of the simplest BlocksWorld programs. Given a block b, remove all of the blocks on top of b. There is one action available, *moveTable*, which has the precondition and outcome desribed in Equations 5.1 and 5.2. The goal G is $\forall x\ \neg(on\ x\ b)$. The starting formula S is Equation 5.3. The BlocksWorld start formula consists of a set of rules that are always true no matter which actions are run. Therefore, at every step, we can always assume S.

HELPS does not depend on the start formula being always true. Before it starts searching, it propagates the start formula through every available action to try to discover which parts of the start formula are universal rules. The algorithm to do this is part of the loop invariant finder and will be described in that section.

The initial partial program $\psi_0 = (S, G)$ with examples $\{\omega_{0,i}\}$ generated from S.

The next partial program is produced by negating the goal $\neg G$ and adding it to the start formula. This produces $\psi_1 = (S \wedge \neg G, G)$ and new examples $\{\omega_{1,i}\}$ are generated for this new starting formula. The next step introduces a loop to solve $\forall x\ \neg(\mathit{on}\ x\ b)$. It creates a set partition $C = x.\neg(\mathit{on}\ x\ b)$. $D = x.(\mathit{on}\ x\ b)$. The new goal, $G_2 = \forall x : C\ \neg(\mathit{on}\ x\ b) \wedge \exists x : D\ \neg(\mathit{on}\ x\ b)$. C and D satisfy their initial definitions at the beginning of the loop body, but their meaning is unknown as soon as the first action is executed. For partial program inside the loop body, C and D are just ordinary sets. The next partial program $\psi_2 = (\{\omega_{2,i}\}, G_2)$. The examples are solution plan states that are determined to be at the beginning of the loop body, as described above.

Since G_2 is a conjunction, a final sub-goal is created to focus attention on one of the conjuncts. The next partial program is $\psi_3 = (\{\omega_{2,i}\}, G_2; \exists x : D\ \neg(\mathit{on}\ x\ b))$. Making an element of D not on b is the final sub-goal. In the next step, the *moveTable* action is considered as a way to make progress on the final sub-goal $\exists x : B\ \neg(\mathit{on}\ x\ b)$. *moveTable* has one parameter term a and two set parameter terms A and B with $A = x.(\mathit{on}\ a\ x)$ and $B = x.\neg(\mathit{on}\ a\ x)$. The outcome of *moveTable* is $\forall x : A\ \neg(\mathit{on}\ a\ x)$. To make progress on the final sub-goal, it must be that $a \in D$ and $b \in A$. The new goal condition is then $\forall x : C\ \neg(\mathit{on}\ x\ b) \wedge (D\ a) \wedge (A\ b) = \forall x : C\ \neg(\mathit{on}\ x\ b) \wedge (D\ a) \wedge (\mathit{on}\ a\ b)$. This is combined with the *moveTable* pre-condition $\forall x\ \neg(\mathit{on}\ x\ a) \wedge \neg(\mathit{on}\ b\ a)$. Combining the pre-condition with the goal condition produces the new open goal:

$$\begin{aligned} G_3 = \forall x : C \quad & \neg(\mathit{on}\ x\ b) \wedge \\ \exists a : D \quad & \forall x\ \neg(\mathit{on}\ x\ a) \\ \wedge \quad & \neg(\mathit{on}\ b\ a) \\ \wedge \quad & (\mathit{on}\ a\ b) \end{aligned}$$

The next partial program is $\psi_4 = (\{\omega_{2,i}\}, G_3)$. The open goal for this partial program G_3 is satisfied by all of the examples. Thus, HELPS hypothesizes that the loop is complete. Now, it tries to find a loop invariant for the hypothesized loop body μ_3. The formula G_3 has the sets C and D instantiated according to their pre-loop-body definitions. We can do this, because G_3 is the formula that must be true at the beginning of each loop iteration. G_3 with C and D taken out is:

$$G_4 = \exists a\ \forall x \neg(\mathit{on}\ x\ a) \wedge \neg(\mathit{on}\ b\ a) \wedge (\mathit{on}\ a\ b)$$

The loop invariant module seeks a formula G_5 such that $G_5 \Rightarrow \tau^{-1}(\mu_3, G_5) \wedge (G_4 \vee \forall x\ \neg(\mathit{on}\ x\ b))$. This circumstance is actually a special case that does not require computing τ^{-1}. HELPS discovers that $S \Rightarrow G_4 \vee \forall x\ \neg(\mathit{on}\ x\ b)$ using deep simplification and the abstract induction theorem prover. Thus, $G_5 = G_4 \vee \forall x\ \neg(\mathit{on}\ x\ b)$. This leads to the last partial program $\psi_5 = (S \wedge \neg G, G_4 \vee \forall x\ \neg(\mathit{on}\ x\ b))$. This partial program is hypothesized to be complete, because its precondition is solved by all examples. $S \Rightarrow G_5$ is then confirmed by the theorem provers. Now, the program is complete, and it is constructed from the individual nodes.

```
μ(ψ5) = ;
μ(ψ4) = ;
μ(ψ3) = μ(ψ2)
μ(ψ2) = (moveTable a) : (D a) ∧ ∀x ¬(on x a) ∧ ¬(on b a) ∧ (on a b)
μ(ψ1) = while ∃x (on x b)
          let C = x : ¬(on x b), D = x : (on x b)
            μ(ψ2)
μ(ψ0) = if ∃x (on x b) then
          μ(ψ1)
```

The outer if statement on $\mu(\psi_0)$ is spurious, but it does no harm. This is the final program. It is provably correct and generated fully autonomously.

5.4.4 Change Propagation

Reasoning about the changes to quantified formulas as they are passed through actions is a critical necessity for HELPS. It is handled by the Change Manager module. Dealing with all of the side-effects and changes is a classical problem in reasoning about plans and programs in situation calculus. It is called the frame problem. In HELPS, due to the precise restrictions placed on parameters and quantified formulas, dealing with change is complicated, yet logical and deterministic.

This section will first show how a quantified formula can be propagated forward and backward through an action. The next sub-section will show how the formula can be propagated backward through a full program with if-then statements and loops. HELPS needs to do forward propagation when it looks for universal rules at the beginning of the synthesis. Thus, it only needs to consider forward propagation through a single action. Backward propagation is an essential component of loop invariant finding, so a backward propagation algorithm through all program types is needed. To illustrate change propagation, a simple example will be considered: propagating the formula $\exists x$ $(on\ x\ b)$ through the *moveTable* action.

Propagating Through Actions

Change propagation operates recursively over a goal formula. First, the outcome of an action is represented as a list of changed set relations. In the case of *moveTable*, the changed relation is $\neg(on\ \{a\}\ A)$. a is the object being moved to the table, and A is the set of objects under a prior to the action. $\kappa(\alpha, \phi)$ denotes the forward change propagation function for action α and quantified formula ϕ. $\kappa^{-1}(\alpha, \phi)$ denotes the backward change propagation. $\kappa(\alpha, .)$, the *action* change manager, is different from $\tau(\mu, .)$, the *program* change manager. Specifically, $\kappa(\alpha, .)$ and $\kappa^{-1}(\alpha, .)$ contain references to the parameters of α, while $\tau(\mu, \phi)$ contains only the sets and terms of ϕ.

κ will now be analyzed in detail. The following basic boolean rules hold:

$$\begin{aligned}
\kappa(\alpha, \phi_1 \wedge \phi_2) &= \kappa(\alpha, \phi_1) \wedge \kappa(\alpha, \phi_2) \\
\kappa(\alpha, \phi_1 \vee \phi_2) &= \kappa(\alpha, \phi_1) \vee \kappa(\alpha, \phi_2) \\
\kappa^{-1}(\alpha, \phi_1 \wedge \phi_2) &= \kappa^{-1}(\alpha, \phi_1) \wedge \kappa^{-1}(\alpha, \phi_2) \\
\kappa^{-1}(\alpha, \phi_1 \vee \phi_2) &= \kappa^{-1}(\alpha, \phi_1) \vee \kappa^{-1}(\alpha, \phi_2)
\end{aligned}$$

The interesting case is when $\phi$is a quantification $Qx : C\ \phi(x)$ for some set C. The change manager considers all of the leaf relations of $\phi(x)$, Each leaf relation ϕ_L has a set relation made by replacing the variables of ϕ_L with the sets of the quantifications from which those free variables came. For example, consider the quantification $\exists x : D\ (on\ x\ b)$. b, as a free variable is constrained by no set. Its set is the ALL set, denoted $\{*\}$. x has the set D. $(on\ x\ b)$ is changed when $x \in \{a\}$ and $b \in A$. Since one can't know whether $b \in A$ or $b \notin A$, the change manager considers both cases as in $\exists x : D\ (on\ x\ b) \wedge \neg(A\ b)\ \vee\ \exists x : D\ (on\ x\ b) \wedge (A\ b)$. When $b \notin A$, $(on\ x\ b)$ is not affected, and $\kappa(\alpha, \exists x : D\ (on\ x\ b) \wedge \neg(A\ b)) = \exists x : D\ (on\ x\ b) \wedge \neg(A\ b)$. On the other hand, when $b \in A$, the formula $\exists x : D\ (on\ x\ b) \wedge (A\ b)$ must be re-written as $(\exists x : D \cap \{a\}\ (on\ x\ b) \vee \exists x : D \cap \neg\{a\}\ (on\ x\ b)) \wedge (A\ b)$. Either $x \in \{a\}$ or $x \notin \{a\}$, so D is partitioned into $D \cap \{a\}$ and $D \cap \neg\{a\}$. The $\exists$ quantification is broken up over this partition in the way described in Section 2.3.

Now, when $x \in \{a\}$ and $b \in A$, $(on\ x\ b)$ is true before the action is run and it is false after the action. So, $\kappa(\alpha, \exists x : D \cap \{a\}\ (on\ x\ b)) = \exists x : D \cap \{a\}\ \neg(on\ x\ b)$. And $\kappa^{-1}(\exists x : D \cap \{a\}\ (on\ x\ b), \alpha) = \exists x : D \cap \{a\}\ F = F$. For comparison $\exists x : D \cap \{a\}\ \neg(on\ x\ b)$, propagated backward, would be

$$\kappa^{-1}(\alpha, \exists x : D \cap \{a\}\ \neg(on\ x\ b) \wedge (A\ b)) = \exists x : D \cap \{a\}\ T = (Exists\ D \cap \{a\})$$

In general, for a relation R, we denote $R \subset R_\alpha$ when the sets of the variables of

R are all subsets of a change relation R_α. When $R \subset R_\alpha$, then $\kappa^{-1}(\alpha, R) = T$, trivial true. $\kappa(\alpha, R) = F$, trivial false, since there is no way that R can hold true before the action is run. If $(\neg R) \subset R_\alpha$, then $\kappa^{-1}(\alpha, R) = F$, and $\kappa(\alpha, R) = \neg R$. Otherwise, $\kappa^{-1}(R, \alpha) = R$ and $\kappa(R, \alpha) = R$. For the sake of symmetry when $(\neg R) \subset R_\alpha$, we set $\kappa(\alpha, R) = T$. This is justified, because we append the outcome of α to $\kappa(\alpha, \phi)$ at the top level and thus avoid losing information. $\kappa_{TOP}(\alpha, \phi) = \kappa(\alpha, \phi) \wedge \alpha_{OUT}$. For *moveTable*, $\alpha_{OUT} = \forall x : A\ \neg(on\ a\ x)$.

Putting these rules together for the example, $\kappa$and κ^{-1}come out to:

$$\begin{aligned}
\kappa(\alpha, \exists x : D\ (on\ x\ b)) = &\quad \neg(A\ b) \wedge \forall x : A\ \neg(on\ a\ x) \wedge \exists x : D\ (on\ x\ b) \\
&\vee\ (A\ b) \wedge \forall x : A\ (\neg(on\ a\ x) \wedge \neg(on\ a\ b)) \wedge \\
&\quad (\exists x : D \cap \neg\{a\}\ (on\ x\ b)) \vee (Exists\ D \cap \{a\}) \\
\kappa^{-1}(\alpha, \exists x : D\ (on\ x\ b)) = &\quad \exists x : D\ (on\ x\ b) \wedge \neg(A\ b) \\
&\vee\ ((\exists x : D \cap \neg\{a\}\ (on\ x\ b)) \wedge (A\ b))
\end{aligned}$$

Propagating Backward Through Programs

κ can model the effects of actions on quantified formulas, but it must leave the parameter terms and parameter sets in the results, because it does not know how these parameters will be instantiated. When a formula is propagated through a program, the output must not have action parameters. However, when a formula is propagated backward through τ^{-1}, it is easy to eliminate these parameters, and so it is possible to compute τ^{-1} for all program types.

First, τ^{-1} will be shown for action calls, $\mu = \alpha(a_1, \ldots, a_k)$. In this case, $\tau^{-1}(\mu, \phi)$ begins with $\kappa^{-1}(\alpha, \phi)$. Since the parameters of α are defined before the action is called, these parameter definitions can be plugged into $\kappa^{-1}(\alpha, \phi)$. The set parameters can be plugged in directly. A parameter term a is defined as some arbitrary term satisfying a quantified formula $\delta(a)$. a can be eliminated from a quantified formula $\gamma(a)$ with the following equation.

$$\gamma_{Elim} = \forall a \; \neg\delta(a) \vee \gamma(a)$$

In words, if $\gamma(a)$ is the formula that must hold on a state ω so that ϕ holds on $\alpha(\omega)$, then any $a \in \omega$ that satisfies the a definition $\delta(a)$ must satisfy $\gamma(a)$. γ_{Elim} reflects this requirement. Another detail concerns free variables of ϕ. All of the possible ways parameter terms could equal free variables must be considered and joined together in a disjunction.

Consider $\phi = \exists x : D \; (on\ x\ b)$ propagated backward through $\mu = (moveTable\ a)$,where a is defined by $(D\ a) \wedge \forall x\ \neg(on\ x\ a) \wedge (on\ a\ b)$. Because of this precondition, $a \neq b$, the free variable. Applying the rules and simplifying,

$$\tau^{-1}(\mu, \phi) \quad = \quad \forall a : D\ (\exists x\ (on\ x\ a) \vee \neg(on\ a\ b) \vee \exists x : D\ (on\ x\ b))$$

This means that for all $a \in D$, if $\forall x\ \neg(on\ x\ a)$ and $(on\ a\ b)$, then there exists $x \in D$ with $x \neq a$ such that $(on\ x\ b)$. In other words, for $\exists x : D\ (on\ x\ b)$ to hold after a *moveTable* call, before the *moveTable* call, there must exist two objects in D that are on b.

Propagating backward through **if-then** statements is straightforward. Let $\mu =$ **if** γ **then** μ_1 **else** μ_2. Then $\tau^{-1}(\mu, \phi) = \gamma \wedge \tau^{-1}(\mu_1, \phi) \vee \neg\gamma \wedge \tau^{-1}(\mu_2, \phi)$.

Propagating backward through a loop is closely related to finding a loop invariant. In fact, the two procedures must call each other recursively. Let $\mu =$ **while**(γ) μ_1. Let $\lambda(\mu_1, \phi \vee \gamma)$ denote the loop invariant for the loop on formula $\phi \vee \gamma$. That is, $\lambda(\mu_1, \phi \vee \gamma)$ is a quantified formula such that $\lambda(\mu_1, \phi \vee \gamma) \Rightarrow \tau^{-1}(\mu_1, \lambda(\mu_1, \phi \vee \gamma)) \wedge (\phi \vee \gamma)$. Thus, we immediately define $\tau^{-1}(\mu, \phi) = \lambda(\mu_1, \phi \vee \gamma)$. At the end of the loop body, either γ holds, which means that the loop will be iterated one more time. Or, ϕ holds. Thus, the equation guarantees that ϕ holds after the loop has run.

Propagating backward through a let environment is another straightforward process. For example, imagine a let consists of a set partition of two sets $A = x.\theta(x)$ and $B = x.\neg\theta(x)$.

$$\begin{aligned}\mu &= \textbf{let } A = x.\theta(x),\ B = x.\neg\theta(x) \\ &\quad \mu'\end{aligned}$$

First the formula ϕ is broken out over the set partition $\{A, B\}$ in the way described in Section 2.3. Then, the broken out ϕ, labeled ϕ', is propagated backward through μ', $\tau^{-1}(\mu', \phi')$. Then, the sets A and B are eliminated by substituting their definitions into $\tau^{-1}(\mu', \phi')$.

Finding Universal Rules

The last use of change management is the finding of universal rules that can be assumed at any time. For example, in BlocksWorld, the rules that hold at the start ($\forall x \forall y\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x)$,etc) hold true no matter what actions are performed. For a formula ϕ, HELPS runs $\kappa(\alpha, \phi)$. Then, it eliminates the parameters by finding the strongest common formula that holds for each parameter partition element. For example, consider $\phi = \forall x \forall y\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x)$ run through the *moveTable* action. κ breaks up ϕ according to the *moveTable* parameter partition $\{\{a\}, A, B\}$,where $A = x : (\mathit{on}\ a\ x)$ and $B = x : \neg(\mathit{on}\ a\ x)$.

$$\begin{aligned}\phi = \quad & \forall x : \{a\}\ \forall y : A\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\ \wedge\ & \forall x : \{a\}\ \forall y : B\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\ \wedge\ & \forall x : A\ \forall y : \{a\}\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\ \wedge\ & \forall x : A\ \forall y : A\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\ \wedge\ & \forall x : A\ \forall y : B\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\ \wedge\ & \forall x : B\ \forall y : \{a\}\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\ \wedge\ & \forall x : B\ \forall y : A\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\ \wedge\ & \forall x : B\ \forall y : B\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x)\end{aligned}$$

Then,

$$\begin{aligned}
\kappa(\alpha, \phi) = \quad & \forall x : \{a\}\ \forall y : A\ \neg(\mathit{on}\ y\ x) \wedge \neg(\mathit{on}\ x\ y) \\
\wedge \quad & \forall x : \{a\}\ \forall y : B\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\
\wedge \quad & \forall x : A\ \forall y : \{a\}\ \neg(\mathit{on}\ x\ y) \wedge \neg(\mathit{on}\ y\ x) \\
\wedge \quad & \forall x : A\ \forall y : A\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\
\wedge \quad & \forall x : A\ \forall y : B\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\
\wedge \quad & \forall x : B\ \forall y : \{a\}\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\
\wedge \quad & \forall x : B\ \forall y : A\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) \\
\wedge \quad & \forall x : B\ \forall y : B\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x)
\end{aligned}$$

Only the occasions when $x \in \{a\}$ and $y \in A$ or $x \in A$ and $y \in \{a\}$ are affected. When $x \in \{a\}$ and $y \in A$, $(\mathit{on}\ x\ y)$ is initially $True$, so $\neg(\mathit{on}\ y\ x)$ must be $True$ at the start and it remains $True$ through the action. Meanwhile, $\neg(\mathit{on}\ x\ y)$ is set to $True$ as $(\mathit{on}\ x\ y)$ is set to $False$. Nevertheless, the strongest inner formula among all partition elements is $\neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x)$. Thus, $\kappa_{Elim}(\alpha, \phi) = \forall x \forall y\ \neg(\mathit{on}\ x\ y) \vee \neg(\mathit{on}\ y\ x) = \phi$. To determine the universal rules, HELPS takes the starting formula S. It initially sets $u = S$. It repeats the action $u = \kappa_{Elim}(\alpha, u) \vee u$ until $\kappa_{Elim}(\alpha, u) \Rightarrow u$. u is then looped through every action so that $\kappa_{Elim}(\alpha, u) \Rightarrow u$ for every available action α. The resulting u is the universal rule.

5.4.5 Finding Loop Invariants

The last major HELPS module to be examined is the loop invariant finder. Consider the loop $\mu = \textbf{while}\ \gamma\ \mu_1$. As discussed before, given a formula ϕ that must hold at the beginning and end of every loop body, the loop invariant finder finds $\lambda(\gamma, \mu_1, \phi)$ such that $\lambda(\gamma, \mu_1, \phi) \Rightarrow (\phi \vee \neg\gamma) \wedge \tau^{-1}(\mu_1, \lambda(\gamma, \mu_1, \phi))$.

The loop invariant finder works by beginning with $\lambda^0 = \phi \vee \neg\gamma$ and iteratively strengthening λ until $\lambda^i \wedge u \Rightarrow \tau^{-1}(\mu_1, \lambda^i)$, where u is the universal rule discussed above. At each iteration, the invariant finder tries to prove $\lambda^i \wedge u \Rightarrow \tau^{-1}(\mu_1, \lambda^i)$. It first tries to generate example worlds satisfying $\lambda \wedge u \wedge \neg\tau^{-1}(\mu_1, \lambda)$. If this fails, it attempts a full-scale proof using deep simplification and abstract induction. The theorem prover either returns success, or it returns a counterexample. So, at each iteration i the invariant finder has a set of all of the counter-examples $\{\nu\}$ seen so far. It then constructs a formula c so that no counter-example ν satisfies c. $c(\nu) = F$. At the next iteration,$\lambda^{i+1} = \lambda^i \wedge c$, so a new counter-example must be found, or λ^i is the returned loop invariant.

To construct c, the invariant finder merges the counter-examples into the simplest, strongest formula, c_{SAT} that satisfies each counter-example ω. c is the negation of this formula. $c = \neg c_{SAT}$. As an example, consider $\phi = \exists a\ \forall x\ \neg(on\ x\ a)$ and $\mu_1 = (moveTable\ a)$. Imagine there was no universal rule, so $u = T$. In this case, initially, $\lambda = \phi$. The first counter-example found is the world $\{x, y\} : \{(on\ x\ y), (on\ y\ x)\}$. In this case, $c_{SAT} = \exists x \exists y\ (on\ x\ y) \wedge (on\ y\ x)$, so $c = \forall x \forall y\ \neg(on\ x\ y) \vee \neg(on\ y\ x)$, the antisymmetric property of *on*. The next counter-example found is

$$\{x, y, z\} : \{(on\ x\ y), (on\ y\ z), (on\ z\ x), \neg(on\ y\ x), \neg(on\ z\ y), \neg(on\ x\ z)\}$$

The counter-example to negate this is $\forall x \forall y \forall z\ \neg(on\ x\ y) \vee \neg(on\ y\ z) \vee \neg(on\ z\ x) \vee (on\ y\ x) \vee (on\ z\ y) \vee (on\ x\ z)$. When this is simplified with respect to the previous anti-symmetry counter-example formula, it becomes $\forall x \forall y \forall z\ \neg(on\ x\ y) \vee \neg(on\ y\ z) \vee \neg(on\ z\ x)$. The next counter-example is $\{(x, y, z, w) : \{(on\ x\ y), (on\ y\ z), (on\ z\ w), (on\ w\ x)\}$. If we continue in the manner we have previously, we could be producing counter-examples forever: all *on* loops of size n, as shown in Figure 5-12. To avoid this circumstance, the invariant finder also has recourse to a set of positive examples, namely the sample instances at the beginning and end of loops.

The positive examples are used to strengthen c such that $c(\nu) = F$ for all counter-examples ν, but such that $c(\omega) = T$ for all positive examples ω. The invariant finder looks for patterns in the counter-examples that are absent in the positive examples. In this way, for the example $\phi = \exists a \forall x \neg(on\ x\ a)$, the transitivity formula

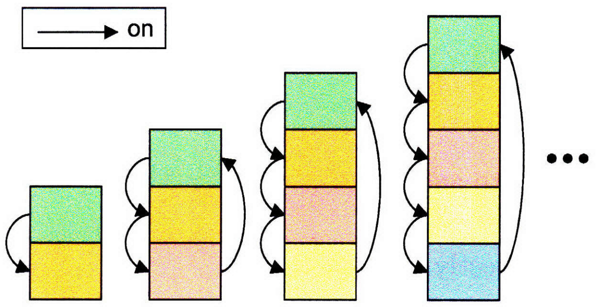

Figure 5-12: A too narrow loop invariant search can yield a never-ending set of counter-examples, like these *on* loops

$\forall x \forall y \forall z\ \neg(on\ x\ y) \vee \neg(on\ y\ z) \vee (on\ x\ z)$ is produced, which is satisfied by all of the positive examples ω and none of the negative examples ν.

With the anti-symmetry axiom and the transitive axiom, the formula λ is finally invariant when passed through the program *moveTable*.

$$\begin{aligned} \lambda = & \quad \exists a \forall x\ \neg(on\ x\ a) \\ & \wedge\ \forall x \forall y\ \neg(on\ x\ y) \vee \neg(on\ y\ x) \\ & \wedge\ \forall x \forall y \forall z\ \neg(on\ x\ y) \vee \neg(on\ y\ z) \vee (on\ x\ z) \end{aligned}$$

In summary, counterexamples are used to make formulas that outlaw precisely the configuration of terms that cause problems with proving the invariant condition. The invariant is repeatedly strengthened until no more counterexamples can be found and $(\phi \vee \neg\gamma) \wedge c \Rightarrow \tau^{-1}(\mu_1, (\phi \vee \neg\gamma) \wedge c$. $(\phi \vee \neg\gamma) \wedge c$ is the returned loop invariant.

5.5 Results and Discussion

5.5.1 Introduction

To test HELPS and guide its development, I have sought to synthesize a variety of generalizations of classic AI problems. There are seven problems in total, described earlier in Section 5.3:

- Clearing a block
- Moving one block onto another
- Moving one block directly onto another
- Putting all blocks into a tower
- Setting a row of switches
- Moving a block into position with a robot arm
- Moving a cursor to a destination over a two-dimensional grid

While the problems are simple to specify, they have proven to be quite difficult to synthesize. The main difficulties are time complexity and never-ending search. Time complexity is an issue that has only been partially resolved. Some of these exercises take hours to finish, and others take days. Problems such as moving a block with an arm have a very large number of cases, and there need to be examples of each case. Otherwise, a program is produced with too narrow pre-conditions. When these pre-conditions are falsified with counter-examples, the whole search must begin again. The more examples used, the slower the system gets, because it must do SATPlan for every example on each iteration of the search. The example problems often require more rounds of counter-examples than I first assumed. HELPS splits into branches for conjunctions and disjunctions. Once a branch is made, the program synthesis proceeds independently for each branch. Because of this, HELPS tends to need examples of nearly every scenario. The lack of generalization across branches is a considerable problem that needs to be addressed in later versions.

The other problem is program length. The more complex a solution program is, the more complex the pre-condition produced at each iteration. Long formulas are considerably more difficult for HELPS to reason about than short ones: deciding sub-goals, proving theorems, propagating backward through loops and actions, etc. For this reason, the recent version of HELPS aggressively seeks to simplify partial program pre-conditions by strengthening them. It strengthens formulas by eliminating components of disjunctions in the formula tree. A strengthened formula is hypothesized as adequate if it is satisfied by the end states of all of the example plans for a given partial program.

Strengthened formulas are essential to keep formula sizes from blowing up in the course of a long search. However, the consequence of formula strengthening is that programs are produced with pre-conditions that narrowly satisfy the particular set of examples that were generated. Thus, HELPS needs to generate more rounds of counter-examples, leading to longer synthesis times.

Even worse than time complexity is the possibility of a never-ending search. Such a search can arise in a variety of ways. It is a natural consequence of working in a high order of logic. Certain partial programs can imply a never-ending sequence of actions. For example, to clear a block a, one must first clear the block directly above a, then move a to the table. HELPS cannot reason about such recursions or prove their termination, so it may produce a never-ending sequence of *moveTable*'s. The lack of reasoning about recursion, and program modularity in general, is a weakness of HELPS that will be addressed in future versions.

Despite these caveats, HELPS can produce programs with a surprising degree of complexity, able to handle a wide variety of input cases. Additionally, the abstract induction theorem prover is often presented with considerably complex theorems to prove in validating a finished program or proving a loop invariant. It has held up in every case thus far, validating the generality of the technique. Certain theorems sometimes take hours to prove, but no theorem has yet been presented that did not terminate. The next sections will examine the BlocksWorld exercise problems in detail.

5.5.2 Clearing a block

Clearing a block is the simplest program to synthesize. There is really only one general case for this problem, and, in the test run, HELPS did not need to get any more rounds of examples. The first hypothesized program was the correct one.

The start formula S is the BlocksWorld formula:

$$\begin{aligned} & \forall x \forall y\ (on\ x\ y) \Rightarrow \neg(on\ y\ x) \\ \wedge\ & \forall x \forall y \forall z\ ((on\ x\ y) \wedge (on\ y\ z)) \Rightarrow (on\ x\ z) \\ \wedge\ & \forall x \forall y \forall z\ ((on\ x\ z) \wedge (on\ y\ z)) \Rightarrow (on\ x\ y) \vee (on\ y\ x) \\ \wedge\ & \forall x \forall y \forall z\ ((on\ z\ x) \wedge (on\ z\ y)) \Rightarrow (on\ x\ y) \vee (on\ y\ x) \end{aligned}$$

S has the free object b broken out of this formula. For example, $\forall x \forall y\ (on\ x\ y) \Rightarrow \neg(on\ y\ x)$ becomes:

$$\begin{aligned} & \forall x \forall y\ (on\ x\ y) \Rightarrow \neg(on\ y\ x) \\ \wedge\ & \forall y\ (on\ b\ y) \Rightarrow \neg(on\ y\ b) \\ \wedge\ & \forall x\ (on\ x\ b) \Rightarrow \neg(on\ b\ x) \end{aligned}$$

The goal is $\forall x \neg(on\ x\ b)$. As developed in the architecture section, the solution, found after only 6 steps is:

```
**if** $\exists x\ (on\ x\ b)$
  **while** $\exists x\ (on\ x\ b)$
    **let** $E = x.\neg(on\ x\ b),\ D = x.(on\ x\ b)$
      **let** $y = y \in D.\ \forall z : E\ \neg(on\ z\ b)\ \wedge \neg(on\ b\ y) \wedge \forall z\ \neg(on\ z\ y)$
        $(moveTable\ y)$;
```

5.5.3 Moving one block onto another

The next BlocksWorld task is, given c and d, to place c onto d. There may be intervening blocks between c and d. The start formula is the same BlocksWorld condition S with the free terms c and d broken out of it. The goal G is (*on c d*). This problem has a surprising amount of complexity. For example d can be on c. It may be directly on c, or there may be intervening blocks between d and c. d may be clear, or there may be other blocks on top of d. Getting the program right in this case took 20 examples, which is three rounds of counter-examples. As with many of the following examples, this program seems incredibly long compared to the program one would write by hand. First, most of the following program is if-then conditions and while conditions. This represents the splitting into various cases, and it typically is the explicit pre-condition to that part of the program. Also, the program divides into many cases that each do similar things. A human programmer would recognize the ability to consolidate these sub-programs, but HELPS does not do so at the moment. The cases have been commented for easy understanding.

Since the solution is complicated, it will be written directly in the list format that HELPS outputs instead of mathematical notation:

```
if !(on c d) {
   // if something is on c or d
  if (Or (Exists z (on z c))
         (on c d)
         (on d c)
         (And (Or (Exists z (And (on d z) (on c z)))
                  (Exists z (on z d)))
              (All y (Or !(on y d)
                         (Exists z (And (on y z) (on c z)))
                         (on d y)
                         (Exists z (on z y))
                         (on c y))))) {
  // if d is either not on c or not directly on c
   if (And !(on c d)
           (Or (And (All z !(on z c))
                  !(on d c)
                  (Or (And (All z (Or !(on d z) !(on c z)))
                           (All z !(on z d)))
                      (Exists y (And (on y d)
                      (All z (Or !(on y z) !(on c z)))
                      !(on d y)
                      (All z !(on z y))
                      !(on c y)))))
              (Exists #1610 (And (on #1610 c) !(on c #1610) !(on d #1610)
                                 (All y !(on y #1610))))
              (And (on d c) (All y !(on y d))))) {
   while (Or (Exists z (on z c))
              (on c d) (on d c)
              (And (Or (Exists z (And (on d z) (on c z)))
                       (Exists z (on z d)))
                  (All y (Or !(on y d) (Exists z (And (on y z) (on c z)))
                             (on d y) (Exists z (on z y)) (on c y))))) {
    let (M=z !(on z c), L=z (on z c)) {
// if something is on c which is clear
      if (And (All y:M !(on y c)) !(on c d)
                (Exists #1610:L (And (on #1610 c) !(on c #1610)
                                     !(on d #1610)
                                     (All y !(on y #1610))))) {
        let (#1610=#1610:L (And (All y:M !(on y c)) !(on c d)
                                (on #1610 c) !(on c #1610)
                                !(on d #1610) (All y !(on y #1610)))) {
          (moveTable #1610);
        }
      }
// if something is on c that is clear and d is not on c
      elseif (And (All y:M !(on y c))
                  !(on c d)
                  !(on d c)
                  (Or (And (All z (Or !(on d z) !(on c z)))
                           (All x !(on x d))
                           (Exists #1135:L (And (on #1135 c)
                                                  !(on d #1135)
                                                  !(on c #1135)
                                                  (All y !(on y #1135)))))
                        (And (Exists a (And (on a d)
                                            (All y (Or !(on a y)
                                                       !(on c y)))
                                            !(on d a)
                                            (All z !(on z a))
                                            !(on c a)))
                             (Exists #1135:L (And (on #1135 c)
                                                  !(on #1135 d)
                                                  !(on d #1135)
                                                  !(on c #1135)
                                                  (All y !(on y #1135))))))) {
          let (#1135=#1135:L (And (All x:M !(on x c))
                                  !(on c d) !(on d c)
                                  (Or (And (All x (Or !(on d x)
                                                      !(on c x)))
                                           (All x !(on x d)))
                                      (Exists a (And (on a d)
                                                     (All x (Or !(on a x)
                                                                !(on c x)))
                                                     !(on d a)
                                                     (All z !(on z a))
                                                     !(on c a))))
                                  (on #1135 c) !(on #1135 d)
                                  !(on d #1135) !(on c #1135)
                                  (All y !(on y #1135)))) {
            (moveTable #1135 K L);
          }
        }
// something is on d and c that is clear and d is on c
        elseif (And (All y:M !(on y c))
                    !(on c d)
                    (Or (And (All z (Or !(on d z) !(on c z)))
                             (Exists #1374:L (And (All x !(on x d))
                                          (on #1374 c)
                                          (on #1374 d)
                                          !(on d #1374)
                                          !(on c #1374)
                                          (All y !(on y #1374)))))
                (And (Exists a (And (on a d)
                                    (All y (Or !(on a y)
                                               !(on c y)))
                                    !(on d a) (All z !(on z a))
                                    !(on c a)))
                     (Exists #1374:L (And (on #1374 c)
                                          (on #1374 d)
                                          !(on d #1374)
                                          !(on c #1374)
                                          (All y !(on y #1374))))))) {
  let (#1374=#1374:L (And (All x:M !(on x c)) !(on c d)
                          (Or (And (All x (Or !(on d x)
                                              !(on c x)))
                                   (All x !(on x d)))
                              (Exists a (And (on a d)
                                             (All x (Or !(on a x)
                                                        !(on c x)))
                                             !(on d a)
                                             (All z !(on z a))
                                             !(on c a))))
                          (on #1374 c) (on #1374 d) !(on d #1374)
                          !(on c #1374)
                          (All y !(on y #1374)))) {
    (moveTable #1374);
  }
}
// if d is on c and d is clear
elseif (And (All y:M !(on y c)) !(on c d) (on d c)
            (All y !(on y d))) {
  (moveTable d);
}
```

```
          }
        }
      }
    // if d is directly on c
      elseif (And !(on c d) (on d c)
                  (Or (And (All y !(on y c))
                           (All y !(on y d)))
                      (Exists #896 (And (on #896 c) (on #896 d) !(on d #896)
                                        !(on c #896)
                                        (All y !(on y #896)))))
                  (All #24698 (Or !(on d #24698) (on c #24698) (on #24698 d)
                                  !(on #24698 c)
                                  (All #24697 (Or (on d #24697) (on c #24697)
                                                  (on #24698 #24697)
                                                  !(on #24697 d) !(on #24697 c)
                                                  !(on #24697 #24698)))))) {
  // if d is not clear
        if (Or (Exists y (on y c)) (on c d) !(on d c) (Exists y (on y d))) {
          while (Or (Exists y (on y c)) (on c d) !(on d c) (Exists y (on y d))) {
            let (F=y (on y c), G=y !(on y c)) {
  // if there is a block on d and c that is clear
              if (And (All y:G !(on y c)) !(on c d) (on d c)
                      (Exists #896:F (And (on #896 c) (on #896 d) !(on d #896)
                                          !(on c #896) (All y !(on y #896))))) {
                let (#896=#896:F (And (All y:G !(on y c)) !(on c d) (on d c)
                                      (on #896 c) (on #896 d) !(on d #896)
                                      !(on c #896) (All y !(on y #896)))) {
                  (moveTable #896);
                }
              }
    // if there is a block on d and c that is clear and directly on d
              elseif (And (All y:G !(on y c)) !(on c d) (on d c)
                          (Exists #655:F (And (All y !(on y d)) (on #655 c)
                                              (on #655 d) !(on d #655)
                                              !(on c #655)
                                              (All y !(on y #655))))) {
```

```
                  let(#655=#655:F (And (All y:G !(on y c)) (All y !(on y d))
                                       !(on c d) (on d c)
                                       (on #655 c) (on #655 d) !(on d #655)
                                       !(on c #655) (All y !(on y #655)))) {
                    (moveTable #655);
                  }
                }
              }
            }
          }
          let (x=d, J=#2982 (on x #2982), K=#2982 !(on x #2982)) {
            (moveTable d);
          }
        }
      }
    // if d is clear, c can be moved directly
      if (And (All z (Or !(on d z) !(on c z))) (All z !(on z c))
              (All z !(on z d)) !(on c d) !(on d c)) {
        (moveBlock c d);
      }
      else {
    // move c to the top element on d
        let (y=y:H (And (on y d) (All z (Or !(on y z) !(on c z)))
                       (All z !(on z c)) !(on d y)
                       (All z !(on z y)) !(on c y)
                       !(on y c))) {
          (moveBlock c y);
        }
      }
    }
```

5.5.4 Putting all blocks in a tower

This program considers taking all of the blocks on the table and putting them in one tower. Since this program has few cases and no free variables, its solution is relatively simple. It may seem complex, but only because the conditions in the if, while, and let statements are rather complex quantified formulas. The start formula S is the standard BlocksWorld beginning. The goal G is $\forall x \forall y\ (on\ x\ y) \vee (on\ y\ x)$. The solution is:

```
if (Exists x (Exists y (And !(on x y) !(on y x)))) {
  while (Exists x (Exists y (And !(on x y) !(on y x)))) {
    let (A=x (Exists y (And !(on x y) !(on y x))),
         B=x (All y (Or (on x y) (on y x)))) {
// #315 is a block which is the bottom of a stack
      let(#315=#315:A
          (Or (And (All y !(on #315 y))
                   (All y !(on y #315))
                   (Exists #406 (And (All z !(on z #406))
                                     (All z:B (All a (Or (on z a)
                                                         (on a z))))
                                     (All z:B (on #315 z))
                                     (All z (Or !(on #315 z) !(on #406 z))))))
              (And (All x:B (All y (Or (on x y) (on y x))))
                   (Exists z (on z #315))
                   (All z !(on #315 z))
                   (All z:B (on z #315))
                   (Exists #406 (And (All z !(on z #406))
                                     !(on #406 #315)
                                     (Exists z (And (All y (Or !(on z y)
                                                              !(on #406 y)))
                                                    !(on #315 z)
                                                    (All y !(on y z))
                                                    !(on #406 z)
                                                    (on z #315)))))))) {
        while (Or (Exists y (And !(on #315 y) !(on y #315)))
                  (Exists x:B (Or (And !(on x #315) !(on #315 x))
                                  (Exists y (And !(on x y) !(on y x)))))) {
          let (E=y (And !(on #315 y) !(on y #315)),
               F=y (Or (on #315 y) (on y #315))) {
// #406 is clear and not on the same stack as #315
            let(#406=#406:E
                 (And (All z !(on z #406)) !(on #406 #315)
                      !(on #315 #406)
                      (All z:B (All a (Or (on z a) (on a z))))
                      (Or (And (All z:B !(on #406 z)) (None F)
                               (All z:B (on #315 z))
                               (All z (Or !(on #315 z) !(on #406 z)))
                               (All z !(on z #315)))
                          (And (All z:F (on z #315)) !(B #406)
                               (All z:B (Or (on z #315) (on #315 z)))
                               (All z:B !(on #406 z))
                               (Exists z:FnA (And (All y (Or !(on z y)
                                                             !(on #406 y)))
                                                  !(on #315 z)
                                                  (All y !(on y z))
                                                  !(on #406 z))))))) {
// if #315 is clear
              if (And (None F) (All z:B (on #315 z))
                      (All z:B !(on #406 z))
                      (All z:B (All a (Or (on z a) (on a z))))
                      (All z (Or !(on #315 z) !(on #406 z)))
                      (All z !(on z #406))
                      (All z !(on z #315))
                      !(on #406 #315) !(on #315 #406)) {
                (moveBlock #406 #315);
              }
// if #315 is not clear, move #406 to the top block on #315
              else {
                let(x=#406,
                    y=y:P (And (All z:F (on z #315))
                               (All z:B (Or (on z #315) (on #315 z)))
                               (All z:B !(on #406 z))
                               (All z:B (on y z))
                               (All z:B (All a (Or (on z a) (on a z))))
                               (Or !(B y) (All a (on y a)))
                               (on y #315) (All z (Or !(on y z) !(on #406 z)))
                               (All z !(on z #406)) !(on #315 y)
                               (All z !(on z y)) !(on #406 y) !(on y #406))) {
                  (moveBlock #406 y);
                }
              }
            }
          }
        }
      }
    }
  }
}
```

5.5.5 Putting one block directly onto another

This is a program very similar to 5.5.3. The goal is $G = (\mathit{on}\ c\ d) \wedge (\forall y\ \neg(\mathit{on}\ c\ y) \vee \neg(\mathit{on}\ y\ d))$. However, this program breaks into even more cases than 5.5.3, because c can be on d with intervening elements, d can be on c, or they can be in different stacks. The solution is:

```
if (Or !(on c d) (Exists x (And (on c x) (on x d)))) {
  // if there is something on d or c
  if (Or (Exists z (on z d)) (Exists z (And (on d z) (on c z)))
         (Exists z (on z c)) (on c d) (on d c)) {
    // if d is on c, or d is not on c and it is clear
    if (And (Or (And (All z !(on z d)) (All z (Or !(on d z) !(on c z)))
                     (All z !(on z c)) !(on c d) !(on d c))
                (And !(on c d)
                     (Or (And (on d c) (All y !(on y d)))
                         (And (Or (Exists #1541396
                                          (And !(on #1541396 d)
                                               !(on d #1541396)
                                          (And !(on d #1541396) (on #1541396 c)
                                               (on #1541396 d) !(on c #1541396)
                                               (All y !(on y #1541396)))))
                              (Or !(on d c)
                                  (And (Exists x (on x d))
                                  (All y (Or (on d y) !(on c y))))))))
           (Or (on d c) (All #261768 (Or !(on #261768 d) (on #261768 c))))) {
     while (Or (Exists z (on z d)) (Exists z (And (on d z) (on c z)))
               (Exists z (on z c)) (on c d) (on d c)) {
       let (B=z (Or !(on d z) !(on c z)), A=z (And (on d z) (on c z))) {
         let (C=z (on z d), D=z !(on z d)) {
           let (F=z !(on z c), E=z (on z c)) {
       // if d is not on c
             if (And (Exists E) !(on d c) (All y:D !(on y d))
                     (All y:F !(on y c)) !(on c d)
                     (Or (And (All y:B (Or !(on d y) !(on c y)))
                              (Exists #1541396:E
                                      (And !(on #1541396 d) !(on d #1541396)
                                           (on #1541396 c) !(on c #1541396)
                                           (All y !(on y #1541396)))))
                         (And (All y:B (Or !(on d y) !(on c y)))
                              (Exists #1541396:E
                                      (And !(on d #1541396) (on #1541396 c)
                                           (on #1541396 d) !(on c #1541396)
                                           (All y !(on y #1541396))))))) {
           // move the top element on c to the table
               let (#1541396=#1541396:E
                    (Or (And (All y:F !(on y c))
                             !(on #1541396 d) (All y:D !(on y d))
                             !(on d #1541396)
                             (All y:B (Or !(on d y) !(on c y))) !(on c d)
                             !(on d c) (on #1541396 c) !(on c #1541396)
                             (All y !(on y #1541396)))
                        (And (All y:F !(on y c)) (All y:D !(on y d))
                             !(on d #1541396)
                             (All y:B (Or !(on d y) !(on c y))) !(on c d)
                             !(on d c) (on #1541396 c) (on #1541396 d)
                             !(on c #1541396) (All y !(on y #1541396))))) {
                 (moveTable #1541396);
               }
             }
```

```
      // if d is on c, it is clear, and there is something under c
           else if (And (Exists A) (on d c) (All y:F !(on y c)) !(on c d)
                        (All y !(on y d))) {
             (moveTable d);
           }
     // if d is on c, and it is clear
           else if (And (on d c) (All y:F !(on y c)) !(on c d)
                        (All y !(on y d))) {
             (moveTable d);
           }
     // if d is on c, and there is something on it
           else {
             let (#1207232=#1207232:E
                  (Or (And (All y:F !(on y c)) !(on #1207232 d)
                           (All y:D !(on y d)) !(on d #1207232)
                           (All y:B !(on c y)) !(on c d) (on #1207232 c)
                           !(on c #1207232) (All y !(on y #1207232)))
                      (And (All y:F !(on y c)) (All y:D !(on y d))
                           !(on d #1207232) (All y:B !(on c y)) !(on c d)
                           (on #1207232 c) (on #1207232 d) !(on c #1207232)
                           (All y !(on y #1207232))))) {
               (moveTable #1207232);
             }
           }
         }
       }
     }
   }
 }
 // if c is on d, or c is not on d and d is not on c
 else if (Or (And !(on d c)
                  (Exists #1260031 (And !(on d #1260031) (on #1260031 d)
                                        !(on c #1260031)
                                        (All y !(on y #1260031))))
                  !(on c d) (All #533948 !(on #533948 c)))
             (And !(on d c) !(on c d)
(Or (Exists #743147
            (And (Or (Exists y (And !(on d y) (on y d)
                                    !(on c y)
                                    (All z !(on z y))
                                    (on #743147 y)))
                     (Exists y (And !(on d y) (on y d)
                                    !(on c y)
                                    !(on #743147 y)
                                    (All z !(on z y)))))
                 (on #743147 c) !(on #743147 d)
                 !(on c #743147) !(on d #743147)
                 (All y !(on y #743147))))
    (Exists #743147
            (And (Or (Exists y (And !(on d y) (on y d)
                                    !(on c y)
                                    (All z !(on z y))
                                    (on #743147 y)))
                     (Exists y (And !(on d y) (on y d)
                                    !(on c y)
                                    !(on #743147 y)
                                    (All z !(on z y)))))
                 (on #743147 c) (on #743147 d)
                 !(on d #743147) !(on c #743147)
                 (All y !(on y #743147))))))
(And (Or (And (All z !(on z d))
              (All z (Or !(on d z) !(on c z)))
              (All z !(on z c)) !(on c d) !(on d c))
         (And !(on d c)
              (Or (And (on c d) (All y !(on y c)))
                  (And
                   (Or (Exists #1260031
                               (And !(on #1260031 c)
                                    !(on d #1260031)
                                    (on #1260031 d)
                                    !(on c #1260031)
                                    (All y !(on y #1260031))))
```

```
                                 (Exists #1260031
                                         (And !(on d #1260031)
                                              (on #1260031 d)
                                              (on #1260031 c)
                                              !(on c #1260031)
                                              (All y !(on y #1260031)))))
                             (Or !(on c d)
                                 (And (Exists x (on x c))
                                      (All y (Or (on c y)
                                                 !(on d y)))))))))
               (Or (on c d) (All #533948 (Or (on #533948 d)
                                             !(on #533948 c)))))) {
   // if c is not on d, and d is not on c
if (Or (And !(on d c)
            (Exists #1260031 (And !(on d #1260031) (on #1260031 d)
                                  !(on c #1260031)
                                  (All y !(on y #1260031))))
            !(on c d)
            (All #533948 !(on #533948 c)))
       (And !(on d c) !(on c d)
            (Or (Exists #743147
                        (And (Or (Exists y (And !(on d y) (on y d)
                                                !(on c y)
                                                (All z !(on z y))
                                                (on #743147 y)))
                                 (Exists y (And !(on d y) (on y d)
                                                !(on c y) !(on #743147 y)
                                                (All z !(on z y)))))
                             (on #743147 c) !(on #743147 d)
                             !(on c #743147) !(on d #743147)
                             (All y !(on y #743147))))
                (Exists #743147
                        (And (Or (Exists y (And !(on d y)
                                                (on y d)
                                                !(on c y)
                                                (All z !(on z y))
```

```
                                                    (on #743147 y)))
                                     (Exists y (And !(on d y) (on y d)
                                                    !(on c y) !(on #743147 y)
                                                    (All z !(on z y)))))
                                (on #743147 c) (on #743147 d)
                                !(on d #743147) !(on c #743147)
                                (All y !(on y #743147))))))) {
// while there is something on c
   while (Or (on d c)
             (All #1260031 (Or (on d #1260031) !(on #1260031 d)
                               (on c #1260031)
                               (Exists y (on y #1260031))))
             (on c d)
             (Exists #533948 (on #533948 c))) {
     let (X=#533948 !(on #533948 c), W=#533948 (on #533948 c)) {
    // move the top element on c to the table
       let (#743147=#743147:W
            (Or (And (All y:X !(on y c)) !(on d c) !(on c d)
                     (Or (Exists y (And !(on d y) (on y d) !(on c y)
                                        (All z !(on z y)) (on #743147 y)))
                         (Exists y (And !(on d y) (on y d) !(on c y)
                                        !(on #743147 y)
                                        (All z !(on z y)))))
                     (on #743147 c) !(on #743147 d) !(on c #743147)
                     !(on d #743147) (All y !(on y #743147)))
                (And (All y:X !(on y c)) !(on d c) !(on c d)
                     (Or (Exists y (And !(on d y) (on y d) !(on c y)
                                        (All z !(on z y)) (on #743147 y)))
                         (Exists y (And !(on d y) (on y d) !(on c y)
                                        !(on #743147 y)
                                        (All z !(on z y)))))
                     (on #743147 c) (on #743147 d) !(on d #743147)
                     !(on c #743147) (All y !(on y #743147))))) {
          (moveTable #743147);
       }
     }
```

```
  }
}
// while there is something on d
while (Or (Exists z (on z d)) (Exists z (And (on d z) (on c z)))
          (Exists z (on z c)) (on c d) (on d c)) {
  let (BD=z (And (on d z) (on c z)), BE=z (Or !(on d z) !(on c z))) {
    let (BF=z (on z c), BG=z !(on z c)) {
      let (BH=z (on z d), BI=z !(on z d)) {
    // if c is not on d
        if (And (Exists BH) !(on c d) (All y:BG !(on y c))
                (All y:BI !(on y d)) !(on d c)
                (Or (And (All y:BE (Or !(on d y) !(on c y)))
                         (Exists #1260031:BH
                                 (And !(on #1260031 c) !(on d #1260031)
                                      (on #1260031 d) !(on c #1260031)
                                      (All y !(on y #1260031)))))
                    (And (All y:BE (Or !(on d y) !(on c y)))
                         (Exists #1260031:BH
                                 (And !(on d #1260031) (on #1260031 d)
                                      (on #1260031 c) !(on c #1260031)
                                      (All y !(on y #1260031))))))) {
       // move the top element on d to the table
          let (#1260031=#1260031:BH
                (Or (And (All y:BI !(on y d)) !(on #1260031 c)
                         (All y:BG !(on y c)) !(on d #1260031)
                         (All y:BE (Or !(on d y) !(on c y))) !(on c d)
                         !(on d c) (on #1260031 d) !(on c #1260031)
                         (All y !(on y #1260031)))
                    (And (All y:BI !(on y d)) (All y:BG !(on y c))
                         !(on d #1260031)
                         (All y:BE (Or !(on d y) !(on c y))) !(on c d)
                         !(on d c) (on #1260031 d) (on #1260031 c)
                         !(on c #1260031) (All y !(on y #1260031))))) {
            (moveTable #1260031);
          }
        }
      // if c is clear and c is on d
          else if (And (on c d) (All y:BI !(on y d)) !(on d c)
                   (All y !(on y c))) {
            (moveTable c);
          }
      // if c is clear and c is on d, and there is something under d
          else if (And (Exists BD) (on c d) (All y:BI !(on y d)) !(on d c)
                       (All y !(on y c))) {
            (moveTable c);
          }
          else {
      // if there is something on d that is clear, move it to the table
            let (#1381120=#1381120:BH
                 (Or (And (All y:BI !(on y d))
                          !(on #1381120 c) (All y:BG !(on y c))
                          !(on d #1381120) (All y:BE !(on d y)) !(on d c)
                          (on #1381120 d) !(on c #1381120)
                          (All y !(on y #1381120)))
                     (And (All y:BI !(on y d)) (All y:BG !(on y c))
                          !(on d #1381120) (All y:BE !(on d y)) !(on d c)
                          (on #1381120 d) (on #1381120 c) !(on c #1381120)
                          (All y !(on y #1381120))))) {
              (moveTable #1381120);
            }
          }
        }
      }
    }
  }
}
// if d is on c (this if statement is spurious)
else if (And (Or (And (All y !(on y d)) (All y !(on y c)) !(on c d)
                      (on d c))
                 (And !(on c d) (on d c)
                      (Or (Exists #1123137 (And !(on #1123137 c)
                                                (on #1123137 d)
```

```
                                                                       !(on d #1123137)
                                                                       !(on c #1123137)
                                                                       (All y !(on y #1123137))))
                                                  (Exists #1123137
                                                          (And (on #1123137 d)
                                                               (on #1123137 c)
                                                               !(on d #1123137)
                                                               !(on c #1123137)
                                                               (All y !(on y #1123137)))))))
                                     (All #106462 (Or !(on #106462 c) (on c #106462)
                                                      (And (Or !(on d #106462) (on #106462 d))
                                                           (Or (on d #106462) !(on #106462 d))
                                                      (All #106463 (Or (on #106462 #106463)
                                                                      !(on #106463 #106462)))))) {
      while (Or (Exists y (on y d)) (Exists y (on y c)) (on c d) !(on d c)) {
        let (CA=y !(on y c), BZ=y (on y c)) {
          let (CB=y (on y d), CC=y !(on y d)) {
            let (#1123137=#1123137:CB
                 (Or (And (All y:CC !(on y d)) !(on #1123137 c)
                          (All y:CA !(on y c)) !(on c d) (on d c)
                          (on #1123137 d) !(on d #1123137) !(on c #1123137)
                          (All y !(on y #1123137)))
                     (And (All y:CC !(on y d))
                          (All y:CA !(on y c)) !(on c d) (on d c)
                          (on #1123137 d) (on #1123137 c) !(on d #1123137)
                          !(on c #1123137) (All y !(on y #1123137))))) {
              (moveTable #1123137);
            }
          }
        }
      }
      (moveTable d);
    }
  }
  (moveBlock c d);
}
```

5.6 Conclusion

This chapter has introduced HELPS, the second of the two types program synthesis systems covered in this thesis. While Spec2Action has the flexibility to change any relation in any order, HELPS uses STRIPS actions as the primitive operation. These actions change many relations at a time, and they can only be used when their preconditions hold. Hence, building programs with STRIPS actions requires careful sequencing of the operations. The resulting programs have loops, conditions, and sequencing, all of the fundamental structures of an iterative program.

HELPS is capable of generating simple programs in the BlocksWorld domain, as well as fundamental programs in other domains, such as navigation and object moving. It generates these programs fully autonomously. That is, there are no special rules, domain theories, or programmer interaction in the program synthesis process. A problem is specified entirely by its starting formula, the primitive actions availble, and the goal formula. To accomplish this, HELPS uses the solutions of examples to guide its synthesis search. It also uses slightly non-traditional representations for logical formulas and primitive actions, which make tractable inductive theorem proving and reasoning about change. With these techniques, HELPS achieves a level of autonomy not realized by previous generalized planning systems that only used deductive synthesis.

HELPS does backward iteration from the goal, refining a partial program at each step. Working backward from the goal, it invokes particular strategies that meet the most important of the goal conditions. The strategies include invoking a primitive action, introducing a loop, and introducing a final sub-goal. To generate provably correct loops, HELPS has special modules to discover loop invariants and to reason about program change.

In the bigger picture, it is my opinion HELPS comes closer than Spec2Action to the ideal of procedural knowledge acquisition that was motivated in the Chapter 1 introduction. Unlike Spec2Action, whose example problems were rather abstract operations on data structures, HELPS solves generalizations of many classical AI problems. However, Spec2Action is able to solve problems that have much more complex goal specifications. Ideally, the two systems should work together. When given an especially difficult goal problem, HELPS should invoke Spec2Action to determine a successful action template. It is then the responsibility of HELPS to determine the proper ordering and control flow of primitive actions to solve this action template. Future extensions, such as combining HELPS with Spec2Action, are discussed in the next chapter, the conclusion.

Chapter 6

Conclusion

This thesis has demonstrated two systems to do automated program synthesis in worlds that are networks of objects and relations. These worlds are a natural and flexible representation for many abstract concepts. So, reasoning about such worlds is an important capability for any autonomous agent that wants to understand how to act at a high level. The primary contribution distinguishing this work from prior work is the use of inductive synthesis to complement deductive program synthesis. By sampling problem instances and generalizing from the instance solutions, these systems have demonstrated an unprecedented level of autonomy.

Form2Grammar is able to deduce a recursive structure for all worlds satisfying a particular quantified formula. It does so by sampling worlds, breaking them down in a consistent way, and hypothesizing a grammar for the breakdowns. This grammar is then automatically proven by a combination of theorem proving techniques. Spec2Action further pursues this idea to generate rather abstract programs to manipulate networks of objects satisfying some start state to satisfy some goal state. It samples examples of the start state, discovers a consistent way to change them to satisfy the goal, and then generalizes from these solutions. It then proves the correctness of the hypothesized program by theorem proving and by automated structural induction over the grammar discovered by Form2Grammar.

HELPS demonstrates the power of inductive and deductive synthesis with a more concrete program type, iterative programs over STRIPS actions. HELPS searches by building partial programs backward from the goal. Its search is guided by example solutions. The example problems are sampled from the starting formula, and they are solved using SATPlan.

The secondary contribution of this thesis are the representation and reasoning algorithms to do deductive reasoning for the two systems and Form2Grammar. The quantified formula knowledge representation assumes that any two objects in a relation are different from each other. Also, in quantified formulas, sets and partitions are first class objects, so that a world is easily split into different classes of objects, and each class can be reasoned about separately. The quantified formula representation is a better representation for reasoning about finite networks of objects and relations than first-order logic, which is better suited for the mathematical reasoning for which it was designed.

In the related work most similar to this thesis, Cresswell has developed a deductive, general planning system that has very similar goals as HELPS. His knowledge representation is linear logic, which is similar to quantified formulas in being very physical and computationally tractable. The broad conclusion is that, for the autonomous problem solving domain, first-order logic is too abstract a knowledge representation, and that there are better, more natural ways to represent general procedural knowledge.

With our quantified formulas, the thesis has shown several novel theorem proving strategies. The abstract induction method can prove quantified formula theorems that would ordinarily require an induction axiom. Such an axiom is not forthcoming in problem definitions. Any proof problem usually only has anti-symmetry and transitivity axioms for certain relation types. So, abstract induction proves theorems in an alternative way. It negates the fact to be proved and shows that no finite world could ever satisfy the formula, or it finds a counterexample.

Another special reasoning technique is partition mapping, which enables structural induction over the grammar produced by Form2Grammar. Given formula P defined over set partition A and a second formula Q quantified over partition B, the partition mapper produces a SAT formula constraining all the ways that A can map to B so that P proves Q. Finally, the constraint that HELPS parameters must fully partition the world set enables very precise and straightforward reasoning about the effects of actions on quantified formulas. This enables HELPS to automatically find loop invariants, which is a formidable task in ordinary program analysis.

From a broder perspective, this thesis has sought to demonstrate that general program synthesis is feasible and can serve as a form of procedural knowledge learning. There are many forms of procedural knowledge learning, from low-level feedback control systems to reinforcement learning. This thesis has shown that it is feasible to do procedural knowledge learning at a high level of abstraction. At such a level, procedural knowledge should be robust and provably correct, so that the knowledge can be applied in the widest range of settings possible.

Programs and subroutines are the best representation for such knowledge, and program synthesis is the process by which such knowledge can be acquired. The two systems, Spec2Action and HELPS, demonstrate that program synthesis can be accomplished using inductive and deductive synthesis, once a problem has been suitably isolated and specified.

There are several directions for future work. First, the capabilities of Spec2Action and HELPS should be combined, and their individual strengths should be applied to solving more complex problems. Second, this abstract procedural knowledge learning should be shown to be a practical for an agent living in a lower-level sensor environment. This means learning the effects of primitive actions and isolating abstract problems out of lower level sensations. For example, a robot in a room should be able to produce a network of objects and relations out of its observation at each (x, y) point of the room. It should learn rules defining which configurations of objects are possible, how its own actions affect this network, and how a concrete visualisation of a goal can be turned into something more abstract.

Bibliography

[1] John R. Anderson. *The Architecture of Cognition.* Harvard University Press, 1983.

[2] John R. Anderson. *The Adaptive character of thought.* Erlbaum, Hillsdale, NJ, 1990.

[3] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[4] Piergiorgio Bertoli, Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Strong planning under partial observability. *Journal of Artificial Intelligence*, 170(4):337–384, 2006.

[5] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, January 2003.

[6] Alessandro Cimatti and Marco Roveri. Conformant planning via symbolic model checking. *Journal of artificial intelligence research*, 159(1-2):127–206, 2004.

[7] Stephen Cresswell, Alan Smaill, and Julian Richardson. Deductive synthesis of recursive plans in linear logic. In *ECP '99: Proceedings of the 5th European Conference on Planning*, pages 252–264, London, UK, 2000. Springer-Verlag.

[8] John Dewey. *Experience and Education.* Simon & Schuster, 1938.

[9] Thomas Ellman. Explanation-based learning: a survey of programs and perspectives. *ACM Computing Surveys*, 21(2):163–221, 1989.

[10] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[11] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[12] Pierre Flener. Achievements and prospects of program synthesis. In *Computational logic: logic programming and beyond, essays in honour of Robert A. Kowalski*, pages 310–346. Springer-Verlag, 2002.

[13] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In *ICAPS*, pages 212–221, 2006.

[14] G. R. Ghassem-Sani and S. W. D. Steel. Recursive plans. In *Proceedings of the European Workshop on Planning EWSP-91*, pages 53–63, St. Augustin, Germany, 1991.

[15] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of IJCAI '69*, pages 219–239. Morgan Kaufmann, 1969.

[16] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Journal of Artificial Intelligence*, 101(1-2):99–134, 1998.

[17] John E. Laird, Allen Newell, and Paul Bloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[18] Hector J. Levesque. Planning with loops. In *IJCAI*, pages 509–515, 2005.

[19] Michael R. Lowry and Jeffrey Van Baalen. Meta-amphion: Synthesis of efficient domain-specific program synthesis systems. *Automated Software Engg.*, 4(2):199–241, 1997.

[20] Z. Manna and R. Waldinger. How to clear a block: A theory of plans. *Journal of Automated Reasoning*, 3(4):343–377, 1987.

[21] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, august 1992.

[22] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings 9th National Conference on Artificial Intelligence (AAAI-91)*, pages 634–639, 1991.

[23] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[24] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[25] Allen Newell and H. A. Simon. *GPS, a program that simulates human thought*, pages 279–293. MIT Press, Cambridge, MA, USA, 1995.

[26] Louise Pryor and Gregg Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[27] Earl D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier, 1977.

[28] Roger Schank. *Dynamic Memory.* Cambridge University Press, 1982.

[29] Roger Schank. *Engines for Education.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1995.

[30] Ute Schmid and Fritz Wysotzki. Induction of recursive program schemes. In *ECML '98: Proceedings of the 10th European Conference on Machine Learning*, pages 214–225, London, UK, 1998. Springer-Verlag.

[31] Ute Schmid and Fritz Wysotzki. Skill acquisition can be regarded as program synthesis: An integrative approach to learning by doing and learning by analogy, 1998.

[32] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[33] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, september 1990.

[34] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM SIGPLAN Notices*, 41(11):404–415, 2006.

[35] Yellamraju V. Srinivas and Richard Jullig. Specware: Formal support for composing software. In *Mathematics of Program Construction*, pages 399–422, 1995.

[36] Siddharth Srivastava. Using abstraction for generalized planning. In *ICAPS*, 2007.

[37] Werner Stephan and Susanne Biundo. Deduction-based refinement planning. In *AIPS*, pages 213–220, 1996.

[38] Gerald J. Sussman. *A Computer Model of Skill Acquisition.* Elsevier Science Inc., New York, NY, 1975.

[39] Jouko Väänänen. A short course on finite model theory.

[40] Roman van der Krogt and Mathijs de Weerdt. Plan repair as an extension of planning. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS-05)*, pages 161–170, 2005.

[41] Elly Winner and Manuela M. Veloso. Distill: Learning domain-specific planners by example. In *ICML*, pages 800–807, 2003.