A MECHANIZED FRAMEWORK FOR SPECIFYING PROBLEM DOMAINS AND VERIFYING PLANS

APPROVED BY

DISSERTATION COMMITTEE:

To my parents

A MECHANIZED FRAMEWORK FOR SPECIFYING PROBLEM DOMAINS AND VERIFYING PLANS

by

SAKTHIKUMAR SUBRAMANIAN, B.Tech., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1993

Acknowledgments

First and foremost, I would like to warmly thank my advisor, Bob Boyer, for everything he did to make this dissertation possible. Apart from lending valuable ideas that helped shape this dissertation, Bob read anything and everything I wrote with the greatest amount of care and always gave me excellent feedback. He was always eager and available for discussing my work even on weekends and holidays not to mention his incredibly prompt response to any electronic mail message that I happened to send him. Thus, it is safe to say that this dissertation would not have materialized without his assistance.

I would also like to thank my committee members Woody Bledsoe, Norman Martin, Ben Kuipers, Bob Simmons and Vladimir Lifschitz for reading drafts of my dissertation and giving useful suggestions. Lifschitz, in particular, brought many subtle technical points to my attention.

There were also a number of others who readily responded to any questions I asked them: Matt Kaufmann, David Russinoff and J Strother Moore of Computational Logic, Inc., Richard Waldinger of SRI and Gordon Novak of UT. They deserve my thanks as well.

Thanks are also due to my friends Yuan Yu, Steve Kaufman, Norm McCain, G. N. Kartha and Hudson Turner for many useful discussions and for commenting on my dissertation.

Finally, I would like to thank my wife, Sandhya, for reading the dissertation and detecting typos, for taking great care of our wonderful son, Amrit, and for extending her love and support even when the going was tough.

The work described here was supported in part by NSF Grant MIP-9017499.

Sakthikumar Subramanian

The University of Texas at Austin
December, 1993

A MECHANIZED FRAMEWORK FOR SPECIFYING PROBLEM DOMAINS AND VERIFYING PLANS

Publication No._______________

Sakthikumar Subramanian, Ph.D.
The University of Texas at Austin, 1993

Supervisor: Robert S. Boyer

This dissertation presents a framework for modeling *problem domains* in the Boyer-Moore logic so that we can verify mechanically *solutions* to various *problems* using the Boyer-Moore theorem prover. A *problem domain* is given by a set of states of the physical world and a set of actions that can be executed sequentially to change state. A *problem* is given by an initial condition and a goal condition. A *solution* is a *plan* that when executed in a state satisfying the initial condition will bring about a goal state. We are mainly interested in verifying plans that involve conditional and repetitive actions for solving problems in domains such as the blocks world. Such domains arise in both artificial intelligence and software engineering.

Our main contribution is a method of specifying problem domains in the Boyer-Moore logic for verifying plans interactively. We illustrate our method by verifying plans for solving problems in some variations of the blocks world. We show how solutions to problems in a class of domains can be verified using the $n \times n$ mutilated checkerboard problem. Our method of specifying domains does not suffer from many of the limitations of current approaches such as the need for stating explicitly a large number of separate frame axioms and state constraints necessary for reasoning about actions with side-effects. Our formalization also allows us to express many other properties of plans such as efficiency requirements. Because both specifications and plans are executable, we can prove properties about them in logic as well as test them on concrete data.

Our method can also be used to obtain a formalization that would enable a program to verify plans depending on changes to domain specifications received as input at various times. Non-monotonic formalisms have generally been used for this purpose but have proven difficult to implement. We illustrate our approach by mechanizing reasoning about actions described in the language $\mathcal{A}$.

Table of Contents

List of Figures

Chapter 1

Introduction

Two important problems are of interest to researchers in both common-sense reasoning in artificial intelligence (AI) and formal methods for software development. One is the problem of verifying whether or not a *plan/program design/prototype* satisfies a *problem specification/requirements specification.* The other is the problem of dealing efficiently, preferably automatically, with incremental changes to problem specifications. This dissertation addresses both of these problems.

To tackle the former, we present a mechanized formal framework in the Boyer-Moore logic for specifying *problem domains* and verifying mechanically, using the Boyer-Moore theorem prover (a.k.a. Nqthm), whether or not a *solution* solves a *problem.* A *problem domain* is defined by a set of possible states of a physical system and a set of actions that can be executed sequentially to change state. A *problem* specifies, in addition, an *initial* condition and a *goal* condition. A *solution* to a problem is a *plan* that when executed in a state satisfying the initial condition brings about a state satisfying the goal. Roughly speaking, a problem domain may specify any sequential physical machine and plans are sequential programs that can be executed to transform an initial state of such a machine to a goal state. Our main contribution is a single mechanized theory for specifying various problem domains and verifying plans, a single programming language-like representation of plans common to all domains and a method of modeling domains conveniently within our framework. We illustrate our approach using problems in two dimensional space such as the $n \times n$ mutilated checkerboard problem and general problems in the blocks world. Our approach gives us a number of advantages. Our method of specifying problem domains and verifying plans does not suffer from many of the limitations of current approaches such as the need for stating explicitly a large number of separate frame axioms and state constraints necessary for reasoning about actions that produce side-effects. Our plan representation has the expressive power of a programming language in that we can express solutions that involve sequencing, conditional execution, iteration and composition of actions. Our formalization also allows us to verify mechanically many other properties of plans such as efficiency requirements.

The problem of minimizing the work to be redone when changes are made to problem specifications is currently being addressed by developing reusable theories in software engineering and by formalized non-monotonic reasoning in artificial intelligence. Since we have a single mechanized framework for formalizing problem domains, theories developed within it are reusable. Non-monotonic formalisms are being developed in AI to enable us to describe to a program the changes to be made to problem specifications but have proven difficult to implement. We show how the needed "non-monotonic" reasoning may be mechanized in the Boyer-Moore logic by including domains as terms in the logic. We illustrate our approach by formalizing the class of finite state machines that can be described in the language $\mathcal{A}$, a language designed primarily for testing non-monotonic formalisms.

1.1 Motivation

The problem of verifying whether or not a plan can bring about a goal state when executed in an initial state arises in both commonsense reasoning [76, 86] and software development [105, 65, 72, 70]. Commonsense reasoning is concerned with the construction of robots that can plan and act to achieve their goals in the physical world. Problem domains in commonsense reasoning and traditional robot planning [30, 2] consist of a robot with preprogrammed action routines that it can execute sequentially to manipulate objects in its physical environment. Consider the well-known example of a robot in a blocks world [37]. There is a table on which there are an arbitrary number of cubical blocks stacked one on top of another. Every block can be either on top of exactly one other block or on the table. There is a robot that can do one of two actions: either move a clear block (one that does not have any other block on top of it) to the top of another clear block or unstack a clear block, i.e., move it to the table. The robot may be given different goals at different times such as "form a stack of 10 blocks" or "put all the blocks on the table". To achieve a goal, the robot must solve the *planning problem* of finding a sequence of actions to transform the current state of the world to a state that satisfies the given goal.

When a robot is given *general problems*, it faces the task of automatically constructing programs. Consider, for example, the problem of clearing a block where we are not told whether the block is already clear or, if not, how many blocks are above it [72]. One plan to solve the problem is to unstack repeatedly the blocks on top of the given block until the latter is clear. Notice that the plan resembles an imperative program in that it involves testing the state of the world and repeatedly doing an action until a desired state is achieved. Since actions are preprogrammed procedures that may be "called" to change the state of the world, a solution to the problem is nothing but a procedure that generates a sequence of calls to the action routines that, when executed, would result in a state in which the given block is clear. Such a procedure has a flavor of a program *design* since it does not depend on the programming language or hardware in which the actions may be implemented. On the other hand, if we view the robot world as a programmable machine whose basic instructions correspond to the atomic actions of the robot, then plans are nothing but programs of such a machine. Observe that the situation is no different even if a human were to program a blocks world robot to clear a given block. The identical programming problem must be solved by designing a suitable procedure. Recognizing the difficulty of generating plans for solving general problems completely automatically, McCarthy and Hayes [86] proposed the construction of a program that can verify plans interactively as a first step towards artificial intelligence[1].

Of course, such problems of program construction using atomic actions arise in various stages of software development. While robot planning involves synthesis of abstract programs from problem specifications for execution by a robot in the external world, *program synthesis* [72, 70, 105, 65], both interactive and automatic, is concerned with the construction of imperative programs from problem specifications for execution by a computer. To see the analogy with planning, consider the following programming exercise.

> Write a PASCAL program that would take a blocks world state and a block as input and produce as output a blocks world state in which the given block is clear. The output state must have the same set of blocks as the input state.

Such a program specification is called a *requirements specification* [63, 64, 16, 20] and is usually the starting point of the *software life cycle* [64, 16]. There are two kinds of specifications usually distinguished during program development, the specification of a program to be constructed before a particular programming language is chosen and the specification of a program to be written in a particular programming language given in terms of the data structures of the programming language [63]. A requirements specification is an informal but precise specification of the program to be constructed couched in the vocabulary of the *customer*, the person who wants the program built. Frequently, a requirements specification involves objects of the physical world [105, 68, 49, 56] as in the blocks world programming exercise. Since customers, more often than not, do not know precisely what their programs must do for them, the requirements specification is obtained by a system designer as a result of *requirements analysis*, which involves studying the problem carefully and ascertaining what is relevant. After the requirements are ascertained, there are one or more design phases whose purpose is to define a program structure consisting of a number of modules that can be implemented independently and that, when implemented, will together satisfy the requirements specification. In general, modules are organized in a hierarchical fashion, with modules higher in the hierarchy implemented in terms of (by means of calls on) modules lower in the hierarchy. For instance, a

[1]It seems unlikely that a problem solving system can generate correct plans to solve general problems automatically since practically any *formal system* [27] such as set theory can be posed as a problem domain with formulas or sequences of formulas as states and inference rules as actions.

design that solves our programming exercise may consist of two procedures: a primitive procedure to unstack a block whenever the block is clear and another procedure that clears a given block by calling the unstack procedure in the manner described above. Although the task is one of modifying representations of blocks world states within a computer, it is clear that a programmer faces the same problem as the robot at the design stage, before a representation in data structures for the world has been chosen viz. the problem of constructing a plan to clear a block using the unstack procedure as an atomic action.

In fact, the problem of combining primitive actions into programs that satisfy an input-output specification arises in practically all stages of top-down, stepwise program development [22, 64, 54]. Jones [54, page 75] describes the process of stepwise program development as follows:

> At each intermediate step of development the given specification is being realized in terms of sub-units. Such a realization is a way of achieving the given specification (documented with pre-/post-conditions) in terms of more basic objects. As such, it consists of specifications (again, using pre-/post-conditions) and a proposed way of combining them. The development process terminates when a step occurs whose realization is achieved entirely in terms of available units. Such units may be the statements of the programming language or use sophisticated services from other support software.

To develop correct programs, we need proofs to show that "given any set of modules which satisfy the specifications of the sub-units, combining such modules in the way specified forms a unit which satisfies the given specification" [54, page 76].

At present, the importance of mechanizing all phases of the software life cycle and the utility of interactive mechanical verification of designs with respect to a specification during the software development process are well understood [52, 105, 65, 53]. The idea is to eliminate errors in the product as early as possible in the software life cycle, preferably well before the program is coded in a programming language. It is now known that the most expensive errors are caused by errors in the requirements specification stemming from a misunderstanding by the designer of the customer's needs [20, 40, 64, 46, 1]. *Formal validation*, the technique of formalizing the requirements specification, proving general properties that follow from the formal requirements specification and then comparing those with the informal requirements specification, has been proposed [40, 46, 33] to minimize such errors. However, in many cases [20, 1], a customer cannot be expected to know exactly what she wants before a *prototype* of the product is built nor be expected to not change the requirements after interacting with the prototype. *Rapid prototyping* [57, 1] has been proposed to minimize the cost incurred when dealing with a changing requirements specification. Here the system designer takes a best guess at a requirements specification and produces a prototype that the customer can interact with. When deficiencies are found, the specification is modified and a prototype is built once again and the cycle repeats until the customer is satisfied with the prototype. The main application that we envisage for our mechanized framework for specifying problem domains and verifying plans is rapid prototyping. Plans for solving problems such as the problem of clearing a block are nothing but prototypes that can be verified against a formal requirements specification given by the initial and goal condition. And this is true whether we are developing programs for computers or for robots (cf. [60]).

Another issue in software development is *program modification* [64, page 252], the changes that must be made to a program perhaps many years after it is written due to changes in the customer's requirements. In such a case, it is useful to minimize the amount of work needed to redesign the program and prove it correct. There are two kinds of modifications to a specification that are possible. One is a change to the problem specification, i.e., changes to the initial and final condition given by a problem. For instance, we may require that space on the table be optimized by putting no more blocks than necessary on the table when clearing a block. Clearly our previous plan to clear a block must be modified to suit the new requirement but the domain theory need not be modified. We may design the following new plan: if the given block b is not clear, then place the topmost block (say x) in its stack on the table. Then repeatedly move the rest of the blocks on top of b to the top of x until b becomes clear. This plan retains the previous procedures for moving and unstacking blocks. It would be nice if all properties about the move and unstack actions that we used to prove the earlier design correct were now available for the proving the correctness of the new design. A more drastic change to the requirements specification is a change to the domain specification. For instance, we may change the blocks world by including other objects such as pyramids and by including more features of the blocks such as weight and color. In such a case, we may have to change our domain specification and, in general, much less of the previous design can be reused for the new design.

The obvious way to deal with the program modification problem is to have an automated reasoning system keep track of previous design histories so that only those parts that need to be changed can be redesigned and proved correct while the rest can be reused from the previous attempts. Thus, reusable theories and libraries [52, 65, 53] have been identified as an important requirement for automated software design. As Woodcock [120] puts it:

> Re-usability is vital to the successful application of formal methods: it allows us to remain flexible by sharing descriptions at every stage of the development process. We are quite used to sharing code—in the form of procedural abstractions in libraries—but here we are talking about

specifications sharing parts, proofs sharing arguments, theories sharing abstractions, even problems sharing common aspects.

Similar sentiments are also expressed in [23, pages 7–8] and [53, pages 9–12].

Even when there are reusable design histories, the burden of figuring out what to redesign and what to reuse rests with the designer. Some work on building automated systems that can take into account changes to problem specifications and programs is currently underway [96, 97]. The idea of constructing a program that can accept as input, declarative facts that describe modifications to problem specifications so that the necessary changes to specifications can be carried out automatically is being pursued by many researchers in commonsense reasoning [79, 77, 81]. For this we need a formalism in which not only all possible domains but also facts describing changes to a domain specification can be expressed. Non-monotonic reasoning [38] has been found necessary for dealing with this problem in AI but has proven extremely difficult to formalize [84]. Gelfond and Lifschitz [35] identified a simple class of domains, a class of finite state systems, which they describe using a language called $\mathcal{A}$ for testing various approaches to formalizing non-monotonic reasoning. We show how the semantics of $\mathcal{A}$ may be mechanized in the Boyer-Moore logic, a first-order logic, by representing all possible domains as terms in the logic.

1.2 A Model Of Problem Domains

For mechanical verification of plans in various domains to be possible, we need the following:

1. A rigorous formalization or model of problem domains.

2. A formal system in which we can formalize or "implement" the models corresponding to problem domains. Our formalization of problem domains must allow us to express and prove "questions" of interest to us. That is, we must be able to express the various problems and solutions that arise in various domains and prove whether or not a solution solves a problem.

3. Since we are interested in a mechanical framework, we need a modeling method that we can use to specify problem domains conveniently. Further, it should be feasible from a practical standpoint to state and prove theorems mechanically. That is, the formulas should not be so long and the proofs so detailed as to render theorem proving practically impossible.

Since we have decided to formalize problem domains in the Boyer-Moore logic and verify plans using the Boyer-Moore theorem prover, points 2 and 3 given above are fairly settled. Thus, we are left with point 1. Fortunately, a rigorous state-transition model of problem domains is already available and, by now, well-understood in AI

[37, 101]. It is also well-known that a fairly large number of problems fit this model [100, 37, 99].

A *problem domain* is given by a set of possible states of a physical world and a set of actions that can be executed sequentially to change the state of the world. Actions are *deterministic* and *terminating*. When an action is executed in a state it results in a unique next state. However, actions, in general, cannot be executed in all states. Every action has associated with it a *precondition*, a predicate that must be satisfied by a state for it to be possible to execute the action in that state[2]. In general, there are an infinite number of states and an infinite number of actual actions (possible state transitions) in a domain but only a finite number of *parameterized* actions. For instance, in the blocks world, there are an infinite number of possible states due to the presence of an infinite number of possible blocks. Similarly, there are an infinite number of particular actions depending on which block or blocks are moved. However, there are only two parameterized actions, move and unstack, which respectively take a pair of blocks and one block as arguments. The precondition of moving block $b1$ to the top of block $b2$ in a state s requires that $b1$ and $b2$ be clear in s and that they be distinct. The precondition of unstacking a block only requires that the block be clear. We allow blocks to be unstacked even when there are no blocks under them.

A particular *problem* in a domain is specified by an initial condition and a goal condition. A *solution* to a problem is a *plan* that when executed in a state satisfying the initial condition will result in a state satisfying the goal. While it is recognized that plans are essentially programs that produce a sequence of actions when executed in a state, at present there seems to be no agreement on a precise representation for plans [90]. Plans for solving general problems must generate distinct sequences of actions depending on the state in which they are executed. Therefore, for the execution of a plan to be possible in a state, it must be possible to execute every action produced by the plan in the intermediate states in which they are executed. For instance, it is possible to clear a block using our plan to repeatedly unstack blocks because every unstack action produced during the execution of the plan is executable.

1.3 Experimental Requirements

The purpose of this section is to outline precisely the experimental requirements of the mechanized theory we are attempting to construct and to discuss some

[2]The use of "precondition" here is somewhat different from preconditions in the program verification literature [24, 54]. In program verification, we usually do not care how the program executes in a state that does not satisfy a precondition but here an action cannot be executed in a state that does not satisfy its preconditions.

of the obstacles we must overcome to achieve our goal.

What are the requirements of a general mechanized framework for modeling problem domains and verifying plans? First of all, since a problem domain can be any sequential physical machine, our method of modeling domains must allow us to specify actions in conventional *von Neumann* machines. However, both states and actions of these machines are considerably simpler than robot worlds such as the blocks world. The states of a robot world may involve arbitrary objects of the physical world and their properties. Also, actions with *side-effects* are more the norm than the exception. By *side-effects*, we mean the changes caused by an action to objects that are *not* mentioned in the action, i.e., to objects that are not its arguments. In programming, side-effects often arise when using pointers. If there are three variables pointing to the same location and one of the variables (an argument of the action) is assigned a new value then it has the side-effect of changing the values of the other two variables. In our blocks world, moving a block $b1$ to the top of another block $b2$ in a state s in which $b1$ is on top of $b3$ produces the side-effect of $b3$ becoming clear. In general, an action may affect an arbitrary number of other objects depending on the state in which it is executed. This would be the case if moving a block caused all the blocks on top of the block being moved to also move along with it to the new location. Actions in the external world may involve creation or destruction of objects including *composite* objects, objects which have other objects as parts. For instance, in the blocks world, unstacking a block that has blocks underneath results in the creation of a new stack. Putting the block back where it belonged results in the destruction of the stack. A similar phenomenon arises in programming when we allow dynamic creation and destruction of data structures. Finally, actions in the external world also have *indirect effects* due to the physical constraints satisfied by the objects in the world. In our blocks world, moving a block from the table to the top of another block causes it to not be on the table, causes the number of stacks to decrease by 1, etc. These indirect effects occur because all states of the blocks world obey the constraint that a block is either on the table or on top of another block but not both.

The idea of a single representation for plans common to all domains seems even more formidable. Such a representation must cover programs that can be expressed by conventional programming languages as well as programs from robot domains. For any given domain (machine), we must be able to express all possible solutions (programs) of the domain. Thus, we need a programming language for each domain such that all possible plans in the domain are the possible programs expressed in the language. Ideally the constructs for describing plans must include most of the familiar programming constructs such as sequencing, conditional execution, repetition and procedural composition.

Apart from being able to express all possible plans, we must be able to express facts involved in proving whether or not a plan solves a problem. For this, we must be able to translate facts about the real world into logical formulas. For example, in order to prove that the plan to clear a block does not change the number of blocks, we must be able to state and prove the invariant that *unstacking a block that exists in any of the infinite number of blocks world states does not change the number of blocks.* To prove that the plan works, we must show that *every block that exists in a state belongs to some stack in the state* and that *unstacking a block $b1$ that is on top of another block $b2$ leaves block $b2$ clear.* Also, we must use mathematical induction to prove that the plan terminates because stacks are of finite length. Since we are interested in a mechanized theory, we must able to state and prove theorems fairly efficiently.

Because our framework for verifying plans mechanically is intended to be used for program development, we must also be able to prove the absence of plans to solve a given problem whenever possible. For instance, if we are asked to write a program to place n queens on an $n \times n$ chessboard so that no queen can take any of the others, we would find out that there are chessboards such as the 2×2 board for which the problem cannot be solved and therefore there does not exist a program to solve the problem. However, carrying out the argument formally using a program-like representation of plans seems to be a difficult task because it requires quantifying over all possible programs. Even more intriguing are proofs that use concepts not mentioned in the problem as in the case of the well-known proof of impossibility of covering a mutilated checkerboard completely with dominoes. Consider an $n \times n$ checkerboard mutilated by removing the squares at the ends of one diagonal. Assume that the squares are not colored. Can such a board be covered completely by placing dominoes such that all the dominoes lie within the board and no two dominoes overlap? It is clear that if n is odd, the $n \times n$ board will have an odd number of squares and so will the mutilated $n \times n$ board. Since placing dominoes results in an even number of covered squares, the mutilated board cannot be covered completely with dominoes. When n is even, there is the following interesting argument [107] involving the use of colors.

> Let us color the squares alternately white and black (as on the usual chess board). The two missing squares have the same color. Thus, the mutilated board has an unequal number of white and black squares. Since each domino covers exactly one black square and one white square, any covering by dominoes covers an equal number of white and black squares and so the desired covering does not exist.

Although the mutilated checkerboard problem was originally posed for *automatic discovery* ([98, 75, 78, 107]) by programs, it is clear that any automated system for interactive verification of program designs must be able to carry out such proofs. Moreover, the problem is not a contrived example: one could easily imagine having to write a graphics program to cover pixels on a screen in the given manner.

Apart from the initial condition and goal condition, a specification may also include other kinds of constraints on solutions such as constraints on efficiency and resource utilization [64]. Normally, we want not just some program that solves a problem but an efficient one. For instance, the plan to clear a block is optimal in the number of steps because every block above the block being cleared must be moved. It would benefit a system designer if he could carry out mechanical proofs about plan efficiency including optimality arguments. Also, programmers sometimes deal with problems that stipulate additional constraints on the solution such as restrictions on the order or the number of times some actions must be executed or restrictions on the properties of states that ensue during the execution of the plan. When designing a program, we may have to take into consideration the available hardware on which the program is going to be ultimately executed to choose our solution. As Dijkstra [22, page 23] puts it:

> It is a programmer's everyday experience that for a given problem to be solved by a given algorithm, the program for a given machine is far from uniquely determined. In the course of the design process he has to select between alternatives; once he has a correct program he will often be called to modify it, for instance because it is felt that an alternative program would be more attractive as far as the demands that the computations make upon the available equipment resources are concerned.

Dijkstra goes on to explain the need for comparing the effects of executing two programs on available equipment in terms of the actions produced when executing them. For instance, at the implementation level, it may be desirable to avoid dynamic creation and deletion of data structures during program execution. Our plan to clear a block creates many new stacks and so we may want to minimize the number of new stacks that are created even though the plan is optimal in the number of actions. Thus, we may want a plan that creates as few new stacks as possible *during* execution. Knowing that unstack actions are the ones that result in new stacks, we may specify this as the constraint that as few unstack actions as possible be used in the solution. Such constraints may also be phrased in terms of properties of states that ensue *during* the execution of the plan. For instance, Flatau [31] discusses the mechanical verification of a compiler that takes into account the resource constraints of the target machine when generating code. The resource constraints take the form of limits on the amount of heap space, control stack space and temporary stack space that a program is allowed to use *during* execution. An analogous constraint on a plan for clearing a block is the restriction that the number of stacks in all the states that ensue during the execution of the plan be no more than 1 + the number of stacks in the initial state. In general, we may want to express arbitrary constraints on the actions that occur in a plan as well as constraints on states that ensue during the execution of a plan or perhaps both. It would be desirable to prove mechanically whether or not a plan satisfies such constraints given in a problem. However, to express many of these theorems, we must be able to express arbitrary predicates on plans.

1.4 Previous Work

We will survey previous work in two parts. The first part relates work done in mechanically generating or verifying program-like plans for solving general problems. We will deal with work done in both artificial intelligence and software engineering. The second part relates work done in dealing with changes to problem specifications.

1.4.1 Plan Verification

McCarthy and Hayes [86] proposed the construction of a proof-checker that can verify interactively whether or not a plan achieves a goal as a first step towards the construction of intelligent robots. The *situation calculus*, a first-order language in which states of the world, actions and plans are explicitly represented as terms, was proposed as a formalism to carry out proofs about plans. The idea that plans or *strategies* could be represented as programs and that program verification techniques could be applied to plan verification was also mentioned. Actions were represented by first-order terms such as $move(A, B)$ and treated as procedure calls in ALGOL. Plans with conditional actions and loops were represented by compound ALGOL statements. There are two main problems in using constructs from a language like ALGOL or PASCAL to model actions. One is the problem of modeling preconditions since actions are *partial functions* executable only in those states that satisfy their preconditions [86, 46]. The other is the problem of specifying actions with side-effects [47].

One of the first attempts to automatically generate recursive plans to solve general problems is that of Manna and Waldinger [69, 72]. The analogy between plans and programs is explained by them as follows[3]:

> Plans are closely analogous to imperative programs in that actions may be regarded as computer instructions, tests as conditional branches and the world as a huge data structure. This analogy suggests that techniques for the synthesis of imperative programs may carry over into the planning domain. Conversely, we may anticipate that insights we develop by looking at a relatively simple planning domain, such as the blocks world, would then carry over to program synthesis in a more complex domain, involving array assignments, destructive list operations, and other alterations of data structures.

[3]While they have observed that imperative programs may also be regarded as plans, we have shown that many problems in the initial stages of the software development process can also be regarded as planning problems (cf. [62]).

They formalize the problem of clearing a block in *plan theory*, a first-order logic based on the situation calculus, and describe the task of mechanically synthesizing a recursive program to solve the problem using their deductive tableau theorem prover.

In plan theory, there are two classes of terms: *static* terms which represent the same entity independent of any state and *fluent* terms that represent entities whose meaning varies from state to state. Static terms include terms that stand for *states*, *objects* such as blocks and table, and *truth values*. Fluent terms include terms such as $hat(d)$ which stands for the object on top of d, $clear(d)$ which stands for the proposition that d is clear and $put(a,b)$ which stands for the atomic action of putting object a on top of object b. The mapping from fluents to the static entities represented by them in various states is formalized using *linkage operators*, :, :: and ;. For example, the static expressions, $s : hat(d)$, $s :: clear(d)$ and $s;put(a,b)$ denote the particular object on top of d in state s, the truth value depending on whether d is clear in state s and the state got by putting a on top of b in s, respectively.

Plans are represented by programs in a Lisp-like programming language using actions as though they were built-in functions. Plans are regarded as fluent terms that may be "linked" to distinct states depending on the properties of the state in which they are executed. For instance, the plan to clear a block mentioned above is represented in plan theory as follows:

$$makeclear(a) \Leftarrow \begin{cases} if & clear(a) \\ then & \Lambda \\ else & makeclear(hat(a)); \\ & put(hat(a), table). \end{cases}$$

Here Λ is an empty plan and ; (an overloaded symbol) stands for functional composition. Unfortunately, the representation of plans as programs without any restriction leads to certain semantic difficulties. To see this, consider the monkey, banana and bomb problem given in [72]. There is a monkey some distance away from two boxes a and b. The monkey is informed that one box contains a banana and the other a bomb but is not told which. His goal is to get the banana without being killed by the bomb. Since terms that stand for various states including future states are represented in the logic, the following program may be produced as a possible solution to the problem.

$$getbanana(s0) \Leftarrow \begin{cases} if & Hasbanana(goto'(s0,a)) \\ then & goto'(s0,a) \\ else & goto'(s0,b). \end{cases}$$

Here $goto'(s0,x)$ is a static term representing the state that would result if the monkey executes the action $goto(x)$ in $s0$. Mathematically speaking, the above "plan" is a valid program. But it is difficult to interpret the program as a sequence of actions to be executed. The "plan" may be read as follows. Go to box a and check if the monkey has the banana. If so, go to box a now, otherwise go to box b. From a physical standpoint, the above program doesn't make sense because it looks at a future state to determine what to do in the present.

Even if we disallow state terms in plans, there is yet another difficulty created by the occurrence of tests in plans. For instance, the following program "solves" the monkey-bomb problem even though it does not include state terms.

$$getbanana(s0) \Leftarrow \begin{cases} if & in(banana, a) \\ then & goto(a) \\ else & goto(b). \end{cases}$$

Here $in(banana, x)$ is a fluent term that is a precondition for the monkey to have the banana on executing the action $goto(x)$. From the problem statement it is clear that there is no single plan that the monkey can execute in all initial states to get the banana. But the above program could be produced as a solution in plan theory if we do not further restrict the possible solutions to the problem. This difficulty arises because we have not taken into account that the monkey cannot test the initial state of the world to determine whether the banana is in a box or not. A similar situation arises with the "bomb in the toilet problem" [88, 116]. To prevent the construction of both forms of non-executable plans as solutions, Manna and Waldinger restrict plans to contain only *primitive* symbols which are described as those symbols "which we know how to execute". While it is clear that the primitivity property of plans is designed to exclude the above kinds of plans, it is unclear whether other operations such as arithmetic operations are allowed to occur in plans. These may be needed, for instance, in a plan that solves the problem of building a tower of n blocks using n clear blocks.

From the standpoint of plan verification, plan theory shares some of the disadvantages commonly associated with the situation calculus[4] style of formalization. First, there is the *frame problem* [86, 69]. In addition to saying what fluents are changed by an action, it is also necessary to provide *frame axioms* which state explicitly what fluents are left unchanged. For example, the primary effect of putting a block on the table is formalized by the following axiom named *put-table-on*:

$$Clear(w, x) \rightarrow On(put'(w, x, table), x, table)$$

for all states w and blocks x. Here $Clear$, On and put' are the static counterparts of the symbols *clear*, *on* and *put* used to form fluent terms. They take an additional

[4]However, the situation calculus is also used for formalizing domains in which the set of possible states and actions are not taken as fixed, unlike the blocks world.

state term as an argument and represent the value of fluents in the given state. The antecedent of the above axiom is the *precondition* of the action and the consequent describes its effect. The axiom says that if x is clear in any state w then it would be on the table in the state got by putting x on the table in w. Notice that the axiom *put-table-on* does not say what happens to fluents other than $on(x, table)$ when the *put* action is executed. To prove, for instance, that the location of a block distinct from x remains unchanged when x is put on the table, we need an additional axiom that stipulates that putting x on the table does not change the locations of blocks distinct from x. What makes the frame problem a problem is that, in domains of realistic complexity, there are many more actions and fluents and so many more frame axioms, thus making it practically impossible to obtain economical problem specifications.

A related problem is the problem of *side-effects* [47, 102, 115]. This precludes the use of the "programming convention" that all fluents other than those explicitly asserted to change in the problem specification are left unchanged by an action. Side-effects are the changes caused by an action in objects other than its arguments. When the action of putting a block a on the table is executed in a state in which a is on top of block b, it also has the side-effect of b becoming clear. But this does not follow from the axiom *put-table-on* nor from frame axioms. Additional axioms are needed because, in general, an action may produce distinct side-effects depending on the properties of the state in which it is executed. The usual approach to tackle side-effects is to include additional axioms such as the *put-table-on* axiom with additional *secondary preconditions* [102].

Yet another difficulty with the situation calculus style of formalization is the problem of determining the *indirect* effects of an action, the so-called *ramification problem*. When a block a is put on the table in a state in which it is on top of another block b, then the action also has the indirect effect of a not being on top of b any more. This fact does not follow from the axioms that describe the direct effects including side-effects of the put action or the frame axioms. The usual solution to deduce indirect effects is to include *state constraints* [37, pages 263–283] as part of the problem specification. For instance, a state constraint that, *in every state a block is either on top of another block or on the table but not both*, would allow us to conclude that a is not on top of b after being unstacked. State constraints are particularly useful when more than one problem must be solved in a domain because they are applicable to all problems.

The additional axioms necessary to describe side-effects of actions and state constraints once again make it difficult to obtain economical problem specifications. The need for a large number of axioms also makes it a problem to create consistent axiomatizations of problem domains. Further, mechanical theorem proving becomes difficult in the presence of a large number of axioms as the following quotation from [72] indicates.

> The reader may have been struck by the complexity of the reasoning required by the *makeclear* derivation, as constrasted with the apparent simplicity of the original planning problem. In fact, the most difficult parts of the proof are involved not with generating the plan itself, but with proving that it meets the specified conditions successfully.

A recent theoretical proposal that also treats plans as programs is reported in [111]. The authors propose a logical framework for specifying consistent axiomatizations of planning domains in Dynamic Logic [45] using a STRIPS-like [30, 29] representation of actions. Unlike the situation calculus, states are not explicitly represented as terms in dynamic logic but are instead referred to using modal operators. The essence of the proposal is to view states as sets of positive ground literals such as $on(A, B)$ and actions and plans as programs that modify states by adding and deleting literals. States are supposed to be axiomatized as *abstract data types* [63] using two operations, an *add* operation and a *delete* operation. The *add* operation adds a literal to a state. The *delete* operation removes a literal from a state if the literal belongs to the state, otherwise it leaves the state unchanged. Essentially, the *add* and *delete* operations play the role of assignment statements in programming languages in that they are the only ones that can be used in plans to modify states. Both actions and plans are programs that use *add* and *delete* operations. Actions do not terminate when executed in states that do not meet their preconditions. Non-termination is modeled using a special instruction **abort** as in [24, 39]. For instance, the action of unstacking a block (putting it on the table) is defined as follows.

```
rec  unstack(x, y).  if     $on(x, y) \wedge clear(x)$
                     then   add − $table(x)$;
                            add − $clear(y)$;
                            delete − $on(x, y)$
                     else   abort fi.
```

Since states cannot be directly mentioned in the language, objects within states are accessed using a *non-deterministic* operator *choose*. *Choose* takes a variable x and a plan as arguments and binds to x an arbitrary object. The operation of *choose* is similar to the parameter passing mechanism for invoking procedures in programming languages: the variable x is bound to a value “guessed” by *choose* in the “environment” in which the plan is executed. Since most recursive plans involve testing the state, they must be defined using the *choose* construct. For instance, our plan to clear a block b requires us to access the topmost block in the stack containing b whenever b is not clear. The plan may be expressed using the *choose* operator as follows.

```
rec $makeclear(x)$.  if    $\neg clear(x)$
                     then  choose   $z$
                           begin    if $clear(z) \wedge above(z, x)$
                                    then $unstack(z)$;
                                    else abort fi
                           end
                     else  skip fi.
```

The frame problem is solved just as in the STRIPS approach. Since *add* and *delete* are the basic operations and we know their complete effects, the complete effects of programs defined using them can also be computed. However, the derivation of frame axioms is not straightforward when the non-deterministic *choose* construct is employed.

The main disadvantage of this approach is the difficulty of expressing actions with side-effects using this STRIPS-like representation of actions [115]. Also, state constraints must be formulated by the user and introduced explicitly as part of the domain specification. In general, it is difficult to formulate constraints in a way consistent with the underlying action semantics. To ensure consistency, the authors propose that domain constraints be proved as *invariants* of actions before they are introduced as part of the domain specification.

So much for relevant research in artificial intelligence. Since von Neumann machines are also problem domains and since plans resemble programs, it behooves us to look carefully into research in mechanical program verification [9, 10, 18]. The work most relevant to this dissertation seems to be the research on formalizing the semantics of individual von Neumann machines such as a microprocessor [11], an assembly language machine [95], a Micro-Gypsy (a Pascal-like language) machine [122] and various other machines [5] in the Boyer-Moore logic for the purpose of verifying programs and systems mechanically. The semantics of these various machines have been formalized by defining Lisp interpreters for various programming languages in the Boyer-Moore logic. Our work differs from these efforts in that it describes a framework in which plans or programs of all domains are expressed using the same programming language. Because we do not have to invent a programming language when specifying a domain, we can formalize much more abstract machines such as the blocks world. Moreover, it does not seem possible to formalize domains such as the blocks world *conveniently* using any of these machines. Many of the programs of (say) a microprocessor do not translate into clear-cut plans in the blocks world and some aspects of the machine state such as program counters or the current instruction do not have a clear-cut meaning in terms of the entities of the blocks world. It is clear that we need a much less detailed programming language whose concepts are closer to the concepts of the physical world for formalizing realistic problem domains. Because we can formalize arbitrary machines within our framework, we hope that this dissertation would enable us to make progress towards the "ideal system verification tool" described by Moore [5, page 410] as follows.

> The ideal system verification tool is nothing more than a general-purpose automated reasoning system: it must be possible to define virtually arbitrary abstract machines and to derive their properties. Such use severely stresses the automated reasoning system because the semantics of interesting machines is extraordinarily complicated.

Another group of researchers [40, 54, 34, 110] in software engineering have proposed *formal specification* languages and techniques for verifying designs of programs with respect to specifications. Three typical specification formalisms [117] are Larch [41], VDM [54] and Z (pronounced "zed") [110]. In all three formalisms, a system (problem domain) to be constructed is typically formalized as a set of states and a set of operations or procedures to change state. In all three formalisms, the state of a system is formalized, as in most imperative programming languages, as a collection of variables associated with values. The variables are allowed to take on values of *abstract data types* that may not be directly available in a programming language. Operations on states are described in terms of the pre- and post-conditions. Unfortunately, the specification of actions with side-effects has been a problem using this approach [63]. A recent paper [7] also explains why the frame problem must be addressed by these specification languages. In any case, mechanical support for developing programs using these formalisms is still at a rudimentary stage [53, 33]. VDM and Z are more notations for humans to use when developing software rather than languages with a precise formal semantics [6]. The Larch Shared Language, however, has been used for expressing and validating some state-independent properties of a specification such as consistency using the Larch Prover [33, 41].

A related approach for developing software is the use of *program transformations* [105, pages 97–192]. However, this approach does not take the problem specification as the starting point of program synthesis. Rather, correctness-preserving transformations are used to synthesize efficient programs from inefficient algorithms that can be easily specified in a very high-level programming language. Yet another approach to program synthesis that is being tried is theorem proving in a *constructive logic* [17]. So far, this approach has been used primarily for constructing *applicative programs* [71] rather than imperative programs.

Artificial intelligence techniques have also been applied to software engineering [105, 65]. However, most systems (e.g. [106, 109]) are of the program transformation kind and do not address the issue of formalizing problem domain specifications. Further, many of the knowledge-based methods [65] do not have facilities for carrying out induction proofs so vital for proving correctness of plans such as the plan to clear a block.

1.4.2 Dealing with Specification Modification

As mentioned before, the most popular approach used by computer scientists in software engineering for dealing with the problem of program modification due to changes to the requirements specification is to use an automated system that maintains reusable theories, libraries and design histories. Some work on building automated systems that can reason about the effects of incremental changes made to specifications and programs is currently underway [96, 97]. A more ambitious approach, that of building a program that can accept as input declarative facts that describe changes to be made to problem specifications is being pursued by many researchers in AI.

The problem was first mentioned by McCarthy [79, 77] and was called the *qualification problem.* One way to think of the qualification problem (in its broadest sense) is as the "robot modification" problem. Normally, when we build a robot like the blocks world robot, we would identify its problem domain constituting its specification and construct it accordingly. If the problem domain (specification) is modified (say) because of the inclusion of new objects like pyramids and their properties or new actions, we would have to rebuild the robot according to the new specification. So much is common to both programming and robot engineering and perhaps most of systems engineering. What AI scientists are looking for is a formalism in which not only all possible domain specifications but also facts about changes to domain specifications can be expressed declaratively. Rather than change an existing domain specification, they want a program to accept facts describing the changes to be made as input and carry out the necessary changes. McCarthy [84] explains this as follows:

> ...the existing formalizations don't have enough of what I call *elaboration tolerance.* The idea is that formalizations should follow human fact representation in being modifiable primarily by extension rather than by replacement of axioms.
>
> Thus, the axioms that allow a program to plan an airplane trip don't take into account either the possibility of losing a ticket or the necessity of wearing clothes. However, a human can modify a plan to take into account either of these requirements should they become relevant but would not revise its general ideas about planning airplane trips to take them into account explicitly. I have an axiomatization of airplane travel that handles losing the ticket nicely, i.e., no axioms about the consequences of losing a ticket or buying a replacement are used in the planning, but if a sentence asserting the loss of a ticket is added, then the original plan can no longer be shown to work and a revised plan involving buying a replacement ticket can be shown to work.

Non-monotonic reasoning has been found necessary for this purpose and rules such as *circumscription* [77, 80, 61] have been proposed for formalizing the needed non-monotonic reasoning.

The ideas behind the problem of specification modification and the need for non-monotonic reasoning are best illustrated using the class of domains (finite state systems) proposed by Gelfond and Lifschitz [35] for testing non-monotonic formalisms. The class of domains under consideration have the following properties:

1. The states of the system are given by the propositional values of a finite number of boolean variables or *fluents*.
2. Actions are not parameterized and are assumed to be executable in all states. However the actual effect of an action when executed in a state may depend on the fluents that are true in the state.

The language $\mathcal{A}$ is used for describing such domains concisely. Each domain is specified by a set of fluents, a set of actions, a set of propositions that specify the effects of actions (e-propositions) and a set of propositions that specify the values of fluents in various situations (v-propositions).

As an example, consider the following Switch Domain described in $\mathcal{A}$. It consists of fluents *Light1* and *Light2*, an action *Switch1* and the following propositions.

$$\begin{array}{l} \textbf{initially } \neg \textit{Light1}, \\ \textbf{initially } \neg \textit{Light2}, \\ \textit{Switch1} \textbf{ causes } \textit{Light1}. \end{array}$$

The first two lines are v-propositions and the last line is an e-proposition. According to the semantics of $\mathcal{A}$, there are 4 states of the Switch Domain depending on the truth values of *Light1* and *Light2*. There is one action *Switch1* which causes *Light1* to become true when executed in any state. Since the semantics of $\mathcal{A}$ incorporates the default rule known as the "commonsense law of inertia" [61, 36] that an action does not affect a fluent unless otherwise mentioned in a domain description, the execution of *Switch1* in any state is assumed by default to leave the value of *Light2* unchanged. Thus, the Switch Domain specifies a simple finite state machine whose states are given by the boolean values of the fluents *Light*1 and *Light2* at various times and whose transitions are given by the actions of the domain viz. *Switch1*. In addition, the actual values of the fluents in various states may be included in a domain description in $\mathcal{A}$. The above domain description says that both *Light1* and *Light2* are false in the initial state.

One form of domain modification that has been frequently considered in formal AI is *addition* of new information to a domain. For instance, we could modify the Switch Domain to the following Extended Switch Domain by adding the following e-proposition:

$$\textit{Switch1} \textbf{ causes } \textit{Light2}.$$

Naturally, this defines a different finite state system and, in general, all facts that are true about the original domain need not remain true about the new domain. For instance, we know that if *Switch1* is executed in a state in which *Light2* is false, then *Light2* will remain false in the Switch Domain but become true in the Extended Switch Domain. The current approach to express domain modifications is by adding formulas that describe a change to a domain to an existing first-order formalization of the domain. However, simply adding formulas describing the new e-proposition to a formalization of the Switch Domain would not give us a formalization of the Extended Switch Domain in classical first-order logic because first-order logic is monotonic: if a formula p can be proved by deduction from a set of formulas Γ then p can also be proved from $\Gamma \cup \{q\}$. Thus, it is felt that a non-monotonic formalism—usually first-order logic augmented by non-monotonic rules such as *circumscription* [77, 80, 61]—is needed to express changes to domain specifications, as the following quotation from Gelfond and Lifschitz [35] indicates.

> The entailment relation of $\mathcal{A}$ is nonmonotonic, in the sense that adding an e-proposition to a domain description D may nonmonotonically change the set of propositions entailed by D. (This cannot happen when a v-proposition is added.) For this reason, a modular translation from $\mathcal{A}$ into another declarative language (that is, a translation that processes propositions one by one) can be reasonably adequate only if this other language is nonmonotonic also.

In a circumscription based formalism similar to [4], (a relevant part of) the Switch Domain may be formalized as follows.

$$\neg noninertial(p, a) \rightarrow (holds(p, s) \equiv holds(p, result(a, s)))$$

$$noninertial(Light1, Switch1)$$

$$holds(Light1, result(Switch1, s))$$

Here variables p, s, and a are universally quantified from the outside over fluents, situations and actions respectively. Circumscription is a rule that is used to minimize the extension of certain predicates common to all domains. By adding facts about these "control" predicates to an existing theory, we can "switch" from a theory of one domain to a theory of another. In the above formalization, the control predicate is *noninertial* which says whether or not an action affects a fluent. The first axiom, sometimes called the "commonsense law of inertia", says that if an action a does not affect a fluent p, then the value of p after a is executed in any situation s

is the same as the value of p in s. We explicitly assert that $Switch1$ affects $Light1$ and specify the effects of $Switch1$ on all fluents that it affects. Since $noninertial$ is minimized via circumscription, $\neg noninertial(Light2, Switch1)$ is provable from the above formulas. Thus, we can prove:

$$holds(Light2, s) \equiv holds(Light2, result(Switch1, s))$$

To obtain the formalization of the Extended Switch Domain without changing the existing axioms, we simply add the following axioms. These clearly invalidate the above inference.

$$noninertial(Light2, Switch1)$$

$$holds(Light2, result(Switch1, s))$$

Thus, we have succeeded in suppressing some of the inferences that are not true in the new domain, i.e., in formalizing non-monotonic reasoning. But as McCarthy [84] explains:

> ...unfortunately the most obvious and apparently natural axiomatizations tend to have unintended models, and this has been observed in several examples, especially the Yale Shooting Problem [42]. This has led to revised formalizations which work but don't seem so natural. It isn't clear whether there is a problem with the systems of nonmonotonic reasoning or whether we simply don't have the right axiom sets.

At present, the formalization of non-monotonic reasoning is a subject of continuing research.

1.5 Our Approach

In this section, we give an overview of our approach, pointing out its advantages and disadvantages. We will first discuss our general framework for mechanizing problem domains and verifying solutions and then explain our approach to formalize "non-monotonic" reasoning.

1.5.1 Specifying Problem Domains

Our formalism is similar to the situation calculus and plan theory in that we model states, actions and plans explicitly as terms in a first-order language. As we have already noted, each problem domain may be thought of as specifying a programmable machine whose basic instructions are the actions of the domain and whose programs are plans. Below we first describe a general method for formalizing the *operational semantics* of such machines in the Boyer-Moore logic so that we can verify mechanically whether or not a plan solves a problem. Using this method, we can formalize the semantics of individual problem domains separately, i.e., each domain is viewed as a separate machine each with its own collection of plans. Later we show how we can generalize these machines into a single machine with a single programming language-like plan representation suitable for all domains.

The states of a problem domain (machine) are modeled by first choosing suitable Lisp data structures to represent them and then defining a Lisp predicate in the logic that recognizes those data structures. This may also require choosing data structures and defining predicates on objects that go into the construction of a state. For instance, we must choose a representation for blocks, define a suitable predicate on blocks and then use it in defining the predicate on blocks world states. Clearly, there are many ways of representing (say) blocks world states. As we shall explain in Chapter 2, the particular choice of data structures to represent states affects the size of the domain specification since it determines the number of explicit state constraints that we must include in the domain specification. The ingenuity of the programmer (i.e. the problem specifier) can make a difference in the number of state constraints that must be explicitly included because properties that must be explicitly asserted about one kind of data structure can be directly proved using the semantics of Lisp when a "better" data structure is chosen. Once a representation of states is chosen, any partial recursive function or predicate on states may be defined in the logic as a suitable Lisp program. For instance, the *on* relation may be defined by a Lisp program that takes blocks *b1* and *b2* and a state *s* as input and returns **t** or **f** depending on whether or not *b1* is on top of *b2* in state *s*.

We distinguish between actions and *specifications* of actions. Roughly speaking, actions are syntactic descriptions consisting of the action name and other parameters whereas the specifications of actions give their input-output behavior and thus are functions from states to states[5]. Thus, the possible actions of a domain are also modeled by defining a Lisp predicate on the data structures that represent them. For instance, the action of moving block $b1$ to the top of block $b2$ may be represented as list (`'move`, *b1*, *b2*). Here *b1* and *b2* are variables and may be substituted by constants such as `'a` and `'b` (that represent blocks) to get `'(move a b)`, the action of moving block `'a` to the top of block `'b`. The specifications of actions are given by Lisp programs which take the input state and the parameters of an action as arguments and return the state got as a result of executing the action in the input state. If the precondition of an action is not satisfied in the input state, an "error state" that is distinct from all legal states of the domain is returned. For instance,

[5]The concepts of actions and specifications of actions are respectively similar in spirit to the concepts of *interface specification* and *behavior specification* used in program specification [63].

the specification of the move action may be defined by a Lisp program res-move (*b1*, *b2*, s) that returns an error state (which we normally represent as a list whose car is `'failed`) if the input state s does not satisfy the preconditions. Otherwise it returns the state got by moving *b1* to the top of *b2* in s. Thus, all the effects of executing an action in a state are specified completely by Lisp programs.

To reason about the effects of executing an action in a state, we define a function *result* that takes an action a and a state s as arguments and returns the state, possibly an error state, got by executing a in s. *Result* may be viewed as a simulator that executes actions by calling the Lisp functions that specify the individual actions with the appropriate parameters. We ensure that the result of executing any action in an error state is the same state. For instance, the *result* function for the blocks world is defined as follows:

```
DEFINITION:
result ($a$, $s$)
=  if car ($s$) = 'failed then $s$
   elseif movep ($a$) then res-move (cadr ($a$), caddr ($a$), $s$)
   else res-unstack (cadr ($a$), $s$) endif
```

Here `movep` is a predicate on move actions and `res-move` and `res-unstack` are Lisp functions that specify the input-output behavior of the move and unstack actions.

Our representation of plans is novel. We represent plans as sequences of actions (straightline programs) where each action may be a primitive action or a *complex action*, where a complex action is an action that is expanded dynamically, at the time of execution, to a sequence of primitive actions by the plan interpreter *resultlist*, depending on the state in which the action is executed. Thus, plans with tests and recursion such as the plan for clearing a block can be represented as complex actions and included in other plans (sequences of actions). We will explain how such plans are represented as complex actions after we present the definition of the interpreter, `resultlist`. `Resultlist` accepts as input a list of actions l and a state s and returns the state got by executing l sequentially in s.

```
DEFINITION:
resultlist ($l$, $s$)
=  if $l \simeq$ nil then $s$
   else resultlist (cdr ($l$), result (car ($l$), $s$)) endif
```

There are two main insights that lead to our representation of recursive plans. Observe, first of all, that it is sufficient to represent plans as *terminating* programs since a plan that solves a problem must transform every state satisfying the initial condition to some state satisfying the goal condition. Thus, the execution of any plan in any state s satisfying the initial condition will only produce a finite sequence of actions, however long, depending on s. (Recall that all actions terminate either in a legal state or in an error state.). Secondly, observe that the sequence of actions produced by a plan when executed in a particular initial state can be computed from the initial state alone; the intermediate states are fully determined by the initial state and the plan to be executed. In fact, for every plan, we can write a Lisp program that would accept the initial state and other parameters of the plan as arguments and return the sequence of actions that the plan will produce when executed in the given initial state. We call such Lisp programs *plan generators* since they are similar to the planning algorithms for generating straightline plans [2, 113] used in AI. For example, the sequence of actions needed to clear a block in various states according to our plan is computed by the following plan generating program `makeclear-gen`. `Makeclear-gen` accepts a block b and a state s as input and returns the finite sequence of actions that would clear b depending on where it is situated in s. In the following definition, `bw-statep` is a predicate on blocks world states and `find-stack-of-block` is a function that accepts a block b and a blocks world state s as arguments and returns the stack (list of blocks) in s to which b belongs or **f** if there is no such stack.

```
DEFINITION:
makeclear-gen ($b$, $s$)
=  if ($\neg$ find-stack-of-block ($b$, $s$)) $\vee$ ($\neg$ bw-statep ($s$))  then f
   elseif $b$ = car (find-stack-of-block ($b$, $s$))  then nil
   else cons (unstack (car (find-stack-of-block ($b$, $s$))),
                 makeclear-gen ($b$,
                                  result (unstack (car (find-stack-of-block ($b$,
                                                                            $s$))),
                                          $s$))) endif
```

Thus, the above recursive Lisp program succeeds in generating the sequence of actions produced by our plan to clear a block using the initial state alone. This means that we need not test the intermediate states that arise during the execution of a plan in order to determine the sequence of actions needed. In fact, `makeclear-gen` may be used to state and prove that our plan to clear a block works as follows:

```
THEOREM: makeclear-works1
 (bw-statep ($s$)
  $\wedge$  find-stack-of-block ($b$, $s$)
  $\wedge$  ($p1$ = makeclear-gen ($b$, $s$))
  $\wedge$  ($s1$ = resultlist ($p1$, $s$)))
 $\rightarrow$  (bw-statep ($s1$) $\wedge$ planp ($p1$) $\wedge$ clear ($b$, $s1$))
```

The above theorem states that for every state s in which there is a stack to which b belongs, the execution of the sequence of actions generated by `makeclear-gen` in s results in a state in which b is clear. In addition, we prove that the sequence of actions generated is a legal sequence. Essentially, we prove that our plan achieves the desired goal by proving that for every state s that satisfies the initial condition, the sequence of actions that would be produced by the plan in s does the job. Thus, we have represented the recursive plan completely using the set of action sequences it would generate in all possible states. This way we do not need constructs for tests and recursion in plans per se. Instead, the constructs for tests and the recursion are embedded in plan generating Lisp programs such as `makeclear-gen`.

Although this does count as verification of the plan, the above method of formalizing problem domains and verifying plans suffers from the following disadvantages.

1. It does not allow us to use the plan to clear a block as a complex action in other plans the way procedures are combined in an imperative programming language. That is, there is no term in the logic that stands for the plan of clearing a block. Such terms would let us describe complex plans succinctly.

2. Using the above method of specifying domains, each domain is formalized as a separate machine with its own collection of actions. Our aim, however, is to obtain a single machine whose states and actions encompass those of every problem domain. The representation of plans of such a general machine may be used uniformly across all problem domains.

We get around these two difficulties by generalizing the representation of actions as follows. Our first step is to represent primitive actions by data structures that stand for the *names* of the functions that specify them. Thus, since the action of moving block x to the top of block y is specified by the function res-move(x, y, s), it is represented by a data structure that stands for $\lambda s.$ res-move(x, y, s) as in [76]. Such terms are called *fluents* by McCarthy [76, 86]. The data structures that represent such lambda expressions are chosen so that `result` can evaluate lambda expressions on particular states using the Lisp interpreter `eval$` available as a function in the logic. `Eval$` (as defined in the logic; see page 92 for details) takes three arguments, *flg*, x and a. If *flg* equals `'list`, `eval$` interprets x as a list of *quotations* (data structures that describe terms syntactically), otherwise it processes x as a quotation of a particular term. The argument a is an "environment" or association list that assigns values to quoted variables, i.e., litatoms. Essentially, `eval$` "unquotes" a quoted term (or list of terms) x and returns its value (or list of values) using the values of the variables specified by a. For instance, given `t`, `'(plus x y)` (the quotation of $x + y$) and an environment `'((x . 2) (y . 3))` as arguments, `eval$` would evaluate x to 2 and y to 3 and apply the definition of `plus` to list (2, 3) to get 5. In the general theory, *result* is defined using `eval$` as follows.

Definition:
result $(a,\ s)$
$=$ **if** car $(s) =$ `'failed` **then** s
else eval\$ (**t**, append $(a$, list (list (`'quote`, $s))),$ **nil**) **endif**

For example, the action of moving `'a` to the top of `'b` is now represented by the data structure `'(res-move 'a 'b)` that stands for λs. res-move (`'a`, `'b`, s). To execute the action `'(res-move 'a 'b)` in a state $s0$, `result` constructs the term corresponding to the description or *quotation* of the term res-move (`'a`, `'b`, $s0$) and evaluates it using `eval$` to obtain the value of res-move (`'a`, `'b`, $s0$). The quotations of terms such as res-move (`'a`, `'b`, $s0$) are already available in the logic because the meta-theory of the Boyer-Moore logic is formalized as part of the logic, i.e., for every term in the logic there exists a term that is its syntactic description or quotation. Notice that we do not represent the bound "lambda variable" explicitly in the representation of actions. Instead we adopt the convention that the last argument of a function that specifies an action is the initial state argument.

This device of representing actions as fluents can be used for representing both primitive and complex actions. For instance, the input-output behavior of the complex action corresponding to the plan given by `makeclear-gen` can be specified by the following function.

Definition:
res-makeclear $(b,\ s) =$ resultlist (makeclear-gen $(b,\ s),\ s)$

`Res-makeclear` is a function that takes a block b and a state s as arguments and returns the state got by executing the plan generated by `makeclear-gen` in s. The complex action corresponding to the plan given by `makeclear-gen` can again be represented as λs. res-makeclear $(b,\ s)$. Since `result` and `resultlist` can once again be used to execute such actions in various states by lambda evaluation, we may allow complex actions to be included in plans. Thus, with our new representation, a plan is a sequence of actions where each action may be either a primitive action or a complex action.

Because all actions of all domains are represented in the same way, we now have a single machine or "programming language" whose states and actions encompass those of every other problem domain. The set of states of this general machine is given by the set of Lisp data structures and the set of actions is given by those data structures that can be interpreted as lambda expressions that are names of action specifications. Plans, sequences of actions, may be thought of as programs of this machine and the interpreter *resultlist* may be thought of specifying the operational semantics. We represent error states of this machine by lists whose car is `'failed`.

Advantages Our method of specifying problem domains for mechanical verification of plans has a number of advantages.

1. Because we specify problem domains by programming, our approach makes it possible to formalize complex domains while retaining the power of logic to prove general properties. Thus, it combines the benefits of *procedural* and *declarative* specifications [76, 92, 119]. The frame problem does not arise any more than when programming a simulator for a domain in Lisp because *all* changes brought about by the execution of an action are specified using a program. Similarly, side-effects are also not a problem because the specifications of actions are Lisp programs that operate on Lisp data structures that represent the entire state, i.e., one that includes all the objects and their properties. Thus, *all* changes to objects that are not arguments to an action can also be carried out on a given state data structure depending on the properties of the objects in it.

2. Since states are modeled as data structures, state constraints—properties true of all states—can be proved from the form of the data structures chosen to represent states. By choosing the state representation cleverly, we can reduce the burden of explicitly stating a large number of state constraints considerably.

3. The problem of building consistent domain specifications is circumvented by adding the programs constituting a domain specification as *definitions* to the logic. This makes domain specifications consistent relative to the Boyer-Moore logic.

4. Our representation of plans as sequences of actions rather than as Lisp-like programs gives us a number of advantages. First, we cannot form "non-executable" plans as in the monkey-bomb problem because neither states nor conditionals occur in the plan representation. Also, any concepts such as arithmetic needed in a plan can be used without any restriction in plan generating programs such as `makeclear-gen`. Our restriction of plans to terminating, straightline programs has the advantage of making plan verification somewhat more tractable than full-blown program verification but has the disadvantage of being inapplicable to domains in which non-terminating plans may be necessary.

5. From the standpoint of rapid prototyping, our approach has the advantage that we can execute requirements specifications a la Lisp programs as well as prove general properties about them as in formal validation [33, 46, 1]. Similarly, whether a particular prototype (plan) satisfies a specification or not can be ascertained by either direct execution on test cases or "scenarios" [64, pages 257–262] or by mechanical verification. The former has the advantage of getting simple bugs out of the way.

6. Because we have a single framework within which all problem domains are formalized, we have the advantage of reusability; work done about one domain can be reused in another. Also, specifications are parameterized and can be composed [14, 33] because they are in a programming language. Since plans are parameterized and can be composed, we allow parameterized and composable designs as well. Both promote reusability.

7. Our formalization also allows us to specify problems with arbitrary constraints including efficiency requirements, restrictions on the types of actions included in a plan as well as constraints on the states that ensue during the execution of a plan.

So far, we have assumed that all the effects of all actions are given in a problem. Sometimes, it may be necessary to specify *partial actions* wherein the effects of actions on only some aspects of a state may be given. In such a case, it would not be possible to define an interpreter for the domain as we have been doing since actions cannot be modeled by programs. We show in Chapter 5 how we may use the `CONSTRAIN` event [8] of Nqthm to specify partial actions. Obviously, we would lose the benefit of executability if actions are specified this way.

1.5.2 Formalizing "Non-monotonic" Reasoning

How do we succeed in formalizing "non-monotonic" reasoning in classical first-order logic? The answer is simple. If we include all possible domains such as the Switch Domain and the Extended Switch Domain as terms in the language, we can express directly in first-order logic that "$Switch1$ does not affect $Light2$ in the $SwitchDomain$" and that "$Switch1$ causes $Light2$ to become true in the $ExtendedSwitchDomain$" without the two facts "interfering" with each other[6]. Here $SwitchDomain$ and $ExtendedSwitchDomain$ are terms in the logic that stand for the Switch Domain and the Extended Switch Domain respectively. This is somewhat similar to the device of introducing explicit state terms that allowed us to express in first-order logic facts that may be true in one situation and false in another without them interfering with each other. Since the effects of executing an action varies from domain to domain, we define the functions `result` and `resultlist` with an extra domain parameter. So we have terms such as result (`'switch1`, *s0*, SWITCH-DOM1) which stands for the result of executing $Switch1$ in state $s0$ of the Switch Domain and prove theorems such as the following:

[6]The connection between this and the proposal to include *contexts* as objects of [81, 85] is unclear. In some sense, different domains are different contexts in which same action may have different meanings.

THEOREM: l-th2
holds (`'(light2 . 0)`, s)
$\rightarrow$ holds (`'(light2 . 1)`, result (`'switch1`, s, SWITCH-DOM2))

The above theorem says that if *Light*2 is false in any state of the Extended Switch Domain (SWITCH-DOM2) then it will become true when *Switch*1 is executed. The inclusion of domains as terms in the logic amounts to formalizing the meta-theory used by designers of non-monotonic rules to prove their correctness and completeness.

Our approach has a number of advantages.

1. Because our formalization is in first-order logic, efficient implementations such as the one given in this dissertation are possible.
2. Since all possible domains are represented as terms in the logic, we can prove very general theorems by defining predicates over domains and by quantifying over domains. We can also define plan generators that output plans as a function of the domain in which the plan is to be executed to solve a problem that arises in more than one domain. Such solutions are applicable to a class of domains.
3. Since proofs of theorems about one domain do not interfere with proofs of theorems about another, we do not need a "truth maintenance" mechanism [25, 21] for deleting proofs that no longer hold simply because the domain has changed.
4. Since domains are represented as data structures in Lisp, we can consider modifications other than addition of new information to a domain. For instance, we could consider deletion of information as a possible modification.

1.6 Contents of the Dissertation

The dissertation is organized as follows.

In Chapter 2, we use our method of specifying problem domains to specify the blocks world we described earlier. We discuss the effect of alternate representations for states on the size of a domain specification and give a number of examples of general problems in the blocks world and plans for solving them. These examples demonstrate how a single domain theory can be reused to verify plans for solving multiple problems within the domain. We also show how a variation of the blocks world (originally used by Fahlman [28] and suggested to us by Richard Waldinger) in which the move action has the side-effect of moving the blocks on top of it can be modeled using our approach. We give an example of a plan to form a single tower using all the blocks in the initial state. The example also demonstrates what happens when changes are made to a domain specification and how theorems common to two similar problem domains can be shared. Plans are expressed as plan generating programs and verified to solve problems. The list of events given as input to the theorem prover to formalize the blocks world and to carry out the mechanical proofs of theorems mentioned in Chapter 2 is in Appendix A.

In Chapter 3, we specify the $n \times n$ mutilated checkerboard problem and describe an interactive mechanical proof of the impossibility of covering a mutilated $n \times n$ board completely with dominoes. The proof formalizes the well-known argument requiring the "invention" of colors. We also show how some plans for covering portions of two dimensional space with dominoes can be expressed. The list of events to formalize the checkerboard domain and carry out the mechanical proof is included in Appendix B.

Chapter 4 describes how solutions to problems in the blocks world that require constraints on the actions composing a plan and on the states generated during the execution of a plan can be expressed using the blocks world formalization of Chapter 2.

Chapter 5 presents our general framework for modeling problem domains and verifying plans. We describe a formalization of a single problem domain or machine whose set of states and set of actions includes those of every other problem domain. Thus, every problem domain can be specified using a subset of states and actions of this general machine. This gives us a uniform way of representing actions and plans of all domains. Also, we show how plans—sequences of actions—can be represented as complex atomic actions and used in other plans much the way procedures are combined in a programming language. We show how the blocks world described in Chapter 2 can be formalized within the general framework and how the mechanical proofs of the theorems in Chapter 2 can be carried out without change using the new formalization. The list of events describing the general framework and a blocks world formalization within the general framework is given in Appendix C.

Chapter 6 deals with the problem of dealing efficiently with modifications to domain specifications. We show that by including domains as terms in the logic, we can describe solutions to problems common to a class of domains. This also avoids the need for non-monotonic reasoning. The example class of domains in this chapter is the class of finite state machines that can described in the language $\mathcal{A}$ [35]. The list of events for formalizing this class of domains and proving typical theorems is included in Appendix D.

In Chapter 7, we summarize the dissertation and suggest directions for future work.

Throughout this dissertation we follow the following style of presentation. Whenever we introduce definitions in the logic for formalizing new concepts and when we state theorems, we usually give an example that "tests" the theorem or definition on concrete Lisp data structures. Apart from enabling the reader to pin down the concepts used in the example, this also demonstrates the suitability of our theory for program development wherein testing is used to get simple errors, particularly in the specification, out of the way. Most of the theorems mentioned in the dissertation have been checked by the theorem prover. Unfortunately, some of them have not been checked mainly due to lack of time. In these cases, we give examples of instantiations of the theorems on concrete data structures that have been proved (automatically) by the theorem prover and then give the statement of the general theorem. We do not feel this is a serious blemish. We believe that the main contribution of this dissertation is the novel representation of plans and the convenient method of programming problem domains that allows us to express theorems needed to be proved in this context as facts about Lisp programs. Since the Boyer-Moore theorem prover is known to prove theorems of the sort mentioned in the dissertation fairly efficiently, it wouldn't be wrong to conclude that mechanical proofs of these theorems are a routine albeit time-consuming exercise. To avoid misunderstanding, we place the dagger symbol shown in the margin of this line alongside theorems that have not † been checked mechanically. We have also explained the difficulties we experienced in getting the theorem prover to check some of the proofs.

We conclude the introduction with a brief review of the automated reasoning system Nqthm, also known as 'the Boyer-Moore Theorem Prover' reproduced with minor alterations from [11].

1.7 The Automated Reasoning System Nqthm

Detailed knowledge of Nqthm is unnecessary for those who are happy enough with the informal paraphrases of the formulas in the remainder of this dissertation. Nqthm is a Common Lisp program for proving mathematical theorems. Since *A Computational Logic* [9] was published in 1979, Nqthm has been used by several dozen users to check proofs of over 16,000 theorems from many areas of number theory, proof theory, and computer science. An extensive partial listing may be found in [10, pages 5–9]. See also [5]. For a thorough and precise description of the Nqthm logic, we refer the reader to the rigorous treatment in [10], especially Chapter 4, in which the logic is precisely defined. In the body of this dissertation, we have been using a conventional syntax rather than the official Lisp-like syntax of Nqthm. The translation between the conventional syntax and the official Lisp-like syntax is reproduced from [12] in Section 1.7.3.

1.7.1 The Logic

The logic of Nqthm is a quantifier-free first order logic with equality. The basic theory includes axioms defining the following:

- the Boolean constants **t** and **f**, corresponding to the true and false truth values.

- equality. $x = y$ is **t** or **f** according to whether x is equal to y.
- an if-then-else function. **if** x **then** y **else** z **endif** is z if x is **f** and y otherwise.
- the Boolean arithmetic operations $x \wedge y$, $x \vee y$, $\neg\, x$, $x \rightarrow y$, and $x \leftrightarrow y$.

The logic of Nqthm contains two 'extension' principles under which the user can introduce new concepts into the logic with the guarantee of consistency.

- *The Shell Principle* allows the user to add axioms introducing 'new' inductively defined 'abstract data types.' Natural numbers, ordered pairs, and symbols are axiomatized in the logic by adding shells:
  - *Natural Numbers.* The nonnegative integers are built from the constant `0` by successive applications of the constructor function 'add1'. The function 'numberp' recognizes natural numbers. The function 'sub1' returns the predecessor of a non-`0` natural number. $x \in \mathbf{N}$ abbreviates numberp(x).
  - *Symbols.* The data type of symbols, e.g., `'failed`, is built using the primitive constructor 'pack' and `0`-terminated lists of ASCII codes. The symbol `'nil`, also abbreviated **nil**, is used to represent the empty list.
  - *Ordered Pairs.* Given two arbitrary objects, the function 'cons' builds an ordered pair of these two objects. The function 'listp' recognizes ordered pairs. The functions 'car' and 'cdr' return the first and second component of such an ordered pair. Lists of arbitrary length are constructed with nested pairs. Thus list($arg_1, \ldots, arg_n$) is an abbreviation for cons(arg_1, ..., cons(arg_n, **nil**)).
- *The Definitional Principle* allows the user to define new functions in the logic. For recursive functions, there must be an ordinal measure of the arguments that can be proved to decrease in each recursion, which, intuitively, guarantees that one and only one function satisfies the definition. Many functions are added as part of the basic theory by this definitional principle. For example, we define for the natural numbers these familiar expressions: $i + j$, $i - j$, $i < j$, $i * j$, $i \div j$, i **mod** j, and exp (i, j). $i \simeq 0$ returns **f** if and only if i is a positive integer. evenp (x) returns **f** if and only if x is an odd positive integer.

The rules of inference of the logic are those of propositional logic and equality with the addition of instantiation and mathematical induction.

1.7.2 The Theorem Prover

Nqthm is a mechanization of the preceding logic. It takes as input a term in the logic, and repeatedly transforms it in an effort to reduce it to non-**f**. Many heuristics and decision procedures are implemented as part of the transformation mechanism.

The theorem prover is fully automatic in the sense that once a proof attempt has started, the system accepts no advice or directives from the user. The only way the user can interfere with the system is to abort the proof attempt. However, on the other hand, the theorem prover is interactive: the system may gain more proving power through its data base of lemmas, which have already been formulated by the user and proved by the system. Each conjecture, once proved, is converted into some 'rules' which influence the prover's action in subsequent proof attempts.

The commands to the theorem prover include those for defining new functions, proving lemmas, and adding shells, etc. The following two commands are the most often used.

- To admit a new function under the definitional principle we invoke:

  DEFINITION: fn-name $(x,\ y)$ = *body*

- To initiate a proof attempt for the conjecture *statement*, naming it lemma-name, we invoke

  THEOREM: lemma-name
  statement

Typically, the checking of difficult theorems by Nqthm requires extensive user interaction. The behavior of the prover is influenced profoundly by the user's actions. The user first formalizes the problem to be solved in the logic. The formalization may involve many concepts and so the specification may be very complicated. The user then leads the theorem prover to a proof of the goal theorem by proving lemmas that, once proved, control the search for additional proofs. Typically, the user first discovers a hand proof, identifies the key steps in the proof, formulates them as a sequence of lemmas, and gets each checked by the prover.

1.7.3 Syntax Summary

Here is a summary of the conventional syntax used in this report in terms of the official syntax of the Nqthm logic described in [10]. ('cond' and 'let' are recent extensions not described in [10].)

1. Variables. x, y, z, etc. are printed in italics.

2. Function application. For any function symbol for which special syntax is not given below, an application of the symbol is printed with the usual notation; e.g., the term `(fn x y z)` is printed as $\mathrm{fn}\,(x,\ y,\ z)$. Note that the function symbol is printed in Roman. In the special case that 'c' is a function symbol of no arguments, i.e., it is a constant, the term `(c)` is printed merely as C, in small caps, with no trailing parentheses. Because variables are printed in italics, there is no confusion between the printing of variables and constants.

3. Other constants. **t**, **f**, and **nil** are printed in bold. Quoted constants are printed in the ordinary fashion of the Nqthm logic, e.g., `'(a b c)` is still printed just that way. `#b001` is printed as $\mathtt{001}_2$, `#o765` is printed as $\mathtt{765}_8$, and `#xA9` is printed as $\mathtt{A9}_{16}$.

4. `(if x y z)` is printed as

   if *x* **then** *y* **else** *z* **endif**.

5. `(cond (test1 value1) (test2 value2) (t value3))` is printed as

   if *test1* **then** *value1* **elseif** *test2* **then** *value2* **else** *value3* **endif**.

6. `(let ((var1 val1) (var2 val2)) form)` is printed as

   let *var1* **be** *val1*, *var2* **be** *val2* **in** *form* **endlet**.

7. The remaining function symbols that are printed specially are described in the following table.

Nqthm Syntax	Conventional Syntax
`(or x y)`	$x \vee y$
`(and x y)`	$x \wedge y$
`(times x y)`	$x * y$
`(plus x y)`	$x + y$
`(remainder x y)`	$x \ \mathbf{mod}\ y$
`(quotient x y)`	$x \div y$
`(difference x y)`	$x - y$
`(implies x y)`	$x \rightarrow y$
`(member x y)`	$x \in y$
`(geq x y)`	$x \geq y$
`(greaterp x y)`	$x > y$
`(leq x y)`	$x \leq y$
`(lessp x y)`	$x < y$
`(equal x y)`	$x = y$
`(not (member x y))`	$x \notin y$
`(not (geq x y))`	$x \not\geq y$
`(not (greaterp x y))`	$x \not> y$
`(not (leq x y))`	$x \not\leq y$
`(not (lessp x y))`	$x \not< y$
`(not (equal x y))`	$x \neq y$
`(minus x)`	$- x$
`(add1 x)`	$1 + x$
`(nlistp x)`	$x \simeq \mathbf{nil}$
`(zerop x)`	$x \simeq 0$
`(numberp x)`	$x \in \mathbf{N}$
`(sub1 x)`	$x - 1$
`(not (nlistp x))`	$x \not\simeq \mathbf{nil}$
`(not (zerop x))`	$x \not\simeq 0$
`(not (numberp x))`	$x \notin \mathbf{N}$

Chapter 2

Plan Verification in the Blocks World

In this chapter, we use our method of formalizing problem domains in the Boyer-Moore logic to specify the blocks world described in Chapter 1. The blocks world has been used as an example problem domain in AI for a long time [118, 112, 28] and at present there is some controversy over whether or not it is a "solved" problem. It is therefore worth emphasizing that we are interested in an implementation that would allow us to express all possible problems and plans, including those that involve conditionals and recursion such as our plan for clearing a block, so that we can verify mechanically whether or not a plan solves a given blocks world problem. While many blocks world implementations exist, very few of them can express general problems and verify plans such as the plan for clearing a block [72] that require mathematical induction. It is perhaps for this reason that Hayes [48] remarked recently that "we do not know how to do the blocks world very well". McCarthy [74] lists the blocks world as an "open problem" and Minsky [93, page 29] justifies the use of the blocks world as follows.

> In attempting to make our robot work, we found that many everyday problems were much more complicated than the sorts of problems, puzzles and games adults consider hard. At every point, in that world of blocks, when we were forced to look more carefully than usual, we found an unexpected universe of problems.

In Section 2.1, we present our formalization of the blocks world. We discuss how the choice of the representation of states in Lisp affects the size of the domain specification by considering alternative representations for blocks world states. In Section 2.2, we give examples of general problems in the blocks world including the problem of clearing a block and show how plans for solving general problems in the blocks world may be expressed as plan generating Lisp programs. We state theorems that must be proved to verify such plans. Our examples demonstrate how the same domain theory can be re-used to solve different problems within a domain. Because the plans use the same set of primitive actions, theorems about the effects of actions proved for one problem can be used for another. In Section 2.3, we specify a variation of the blocks world that shows more clearly how our method is applicable to domains in which actions have side-effects. This blocks world [28] includes exactly one move action which has the side-effect of moving all the blocks above the block that is moved along with it. We present an example of a plan to form a single tower using all the blocks in the initial state. The example also demonstrates what happens when changes are made to a domain specification and how theorems common to two similar problem domains can be shared.

The list of events given as input to the theorem prover to formalize the blocks world and prove the theorems mentioned in this chapter are included in Appendix A. Throughout the chapter, we show how theorems can be "tested" by stating instantiations of theorems with variables substituted by concrete data structures. Apart from demonstrating that both the specifications and theorems can be tested on "scenarios", the examples are also meant to help the reader grasp the actual representations used in our formalization. The proofs of such concrete theorems were carried out automatically by the theorem prover.

2.1 Formalization of the Blocks World

To define the interpreter *resultlist* for blocks world plans, we must choose data structures to represent blocks world states and actions. Since we want to express general theorems, we need predicates on states, actions and plans. The predicate on states must enable us to prove various state constraints, properties true of all states, needed for verifying solutions to various problems. As mentioned earlier, the particular choice of data structures to represent states influences the number of state constraints that we must explicitly assert when defining the predicate on states. We illustrate this aspect by trying out two different representations of blocks world states. Our first representation of blocks world states is as a list of terms that stand for primitive relations between blocks in a state such as $On(A, B)$, $Ontable(D)$ and $Clear(C)$. This representation has been used in AI programs written in PLANNER [118] and Prolog [58], and is used currently by many existing planning programs [113]. The representation turns out to be inconvenient for specifying the predicate on the set of possible states because it forces us to specify a number of state constraints explicitly. By choosing a different representation that allows state constraints to be directly deduced from the semantics of Lisp data structures a more tractable formalization is obtained.

2.1.1 A First Attempt

In many contemporary planners [73, 94], states are represented by a list of terms that stand for primitive relations between blocks such as $On(A, B)$, $Ontable(D)$ and $Clear(C)$. Let us see what it takes to define a predicate on blocks world states using this representation. We may represent a blocks world state in Lisp as a list of pairs of litatoms. All litatoms other than `'table` and `'clear` stand for blocks. The litatom `'table` stands for the table. If a pair cons (`'clear`, x) belongs to state s, then x is clear in s. If a litatom pair cons $(x,\ y)$, where x and y are distinct from `'clear`, belongs to a state s, then x is on top of y in s. An example of a state is `'((clear . a) (a . c) (c . table) (clear . b) (b . table))`. In this state there are blocks `'a`, `'b` and `'c` such that `'a` is clear and on top of `'c` which is on the table, and `'b` is on the table and clear.

Now, let us try to define a predicate on states using this representation. For this we need the following predicate on blocks.

DEFINITION:
blockp (x) = (litatom (x) $\wedge$ $(x \neq$ `'table`$)$ $\wedge$ $(x \neq$ `'clear`$)$)

For a list of litatom pairs to be a legal blocks world state, it must satisfy the following constraints. The litatom `'clear` should not occur as the cdr of any pair in the list and the litatom `'table` should not occur as the car of any pair in the list. Further, the car of no pair belonging to a state should be equal to its cdr. These constraints may be expressed as follows using the predicate `pairset`.

DEFINITION:
```
pairset (s)
=  if s ≃ nil then t
   else litatom (caar (s))
        ∧ litatom (cdar (s))
        ∧ (caar (s) ≠ 'table)
        ∧ (cdar (s) ≠ 'clear)
        ∧ (caar (s) ≠ cdar (s))
        ∧ pairset (cdr (s)) endif
```

This rules out states such as `'((clear . clear) (table . a))`. But we must further ensure that every block (that exists in a state) belongs to exactly one stack. That is, no block can be immediately above or below more than one object. This constraint can be met by ensuring that every block occurs as a car in exactly one litatom pair in a state and that every block occurs as a cdr in exactly one litatom pair in a state. Our definition of `pairset` ensures that a block cannot be the car and the cdr of the same litatom pair. The remaining constraints can be specified using the following predicates.

DEFINITION:
```
check-block (x, s)
=  if s ≃ nil then t
   elseif x = cdar (s) then f
   else check-block (x, cdr (s)) endif
```

```
DEFINITION:
no-two-blocks-on-top-of-block ($s$)
=  if $s \simeq$ nil then t
   else ((¬ blockp (cdar ($s$))) ∨ check-block (cdar ($s$), cdr ($s$)))
        ∧  no-two-blocks-on-top-of-block (cdr ($s$)) endif
```

```
DEFINITION:
block-not-on-two-blocks ($s$)
=  if $s \simeq$ nil then t
   else ((¬ blockp (caar ($s$))) ∨ (¬ assoc (caar ($s$), cdr ($s$))))
        ∧  block-not-on-two-blocks (cdr ($s$)) endif
```

The above definitions rule out states such as `'((a . table) (a . c))`, and `'((a . b) (c . b))` but still don't get the job done. Lists in which there are "hanging" chains of blocks that do not rest on the table such as `'((a . b) (b . c))` are not covered by the above predicates. Stating these constraints would require still additional predicates which do not appear to be definable concisely. It is worth noting that, with the chosen representation, it takes a lot of effort even to figure out what extra constraints are needed in addition to the effort needed to express them succinctly as Lisp programs.

2.1.2 Our Representation

On the other hand, we may represent a blocks world state as a set of stacks where each stack is a finite, non-empty sequence of distinct blocks. The stacks in a state are pairwise disjoint. The first block of a stack is clear and the last block is on the table. We represent blocks as litatoms and stacks as sequences of distinct litatoms. Thus, the blocks world state given in the figure on the next page in which there are two stacks, one with blocks `'a`, `'b` and `'c` and the other with blocks `'d` and `'e` is represented by STATE1.

DEFINITION: STATE1 = `'((a b c) (d e))`

The predicates `blockp` and `stackp` respectively recognize blocks and stacks. It turns out to be convenient from a theorem proving standpoint to force the cdr of the last block in a stack to be **nil** rather than allow it to be equal to any non-list such as numbers or other litatoms.

DEFINITION: blockp (x) = litatom (x)

Figure 2.1: A blocks world state

```
DEFINITION:
stackp (l)
=  if $l \simeq$ nil then f
   elseif cdr (l) = nil then blockp (car (l))
   else blockp (car (l))
        $\wedge$ (car (l) $\notin$ cdr (l))
        $\wedge$ stackp (cdr (l)) endif
```

The predicate `disjoint` checks if the two given lists are disjoint and `disjointlist` checks if a given list $s1$ is disjoint with every list in $l1$.

```
DEFINITION:
disjoint (l1, l2)
=  if $l1 \simeq$ nil then t
   else (car (l1) $\notin$ l2) $\wedge$ disjoint (cdr (l1), l2) endif
```

```
DEFINITION:
disjointlist (s1, l1)
=  if $l1 \simeq$ nil then t
   else disjoint (s1, car (l1)) $\wedge$ disjointlist (s1, cdr (l1)) endif
```

We can now define a blocks world state as a set of stacks. Once again we force the cdr of the last element to be **nil**.

```
DEFINITION:
bw-statep (x)
=  if $x \simeq$ nil
   then if x = nil then t
        else f endif
   else stackp (car (x))
        $\wedge$ disjointlist (car (x), cdr (x))
        $\wedge$ bw-statep (cdr (x)) endif
```

The following theorem tests the above definition on STATE1.

THEOREM: blocks-world-example1
bw-statep (STATE1)

Notice that many of the constraints that we had to provide explicitly using our earlier representation can now be proved as theorems using properties of Lisp data structures. For instance, it is clear from our Lisp representation that there is at most one block immediately above and immediately below every block in a state simply because there is at most one element immediately preceding and immediately succeeding a member of a list. Similarly, we can prove using properties of lists that there are no "hanging" chains of blocks because we know that every finite, non-empty list has a last element which is defined to be on the table. Thus, a "good" representation of states is one that allows us to prove state constraints using the available axioms about Lisp data structures.

Since we have all possible states as data structures, we may now define any partial recursive function or predicate on states as a Lisp program. Below we define predicates `on`, `ontable` and `clear`. A block is on the table in a state if there is a stack in the state of which it is the last element. The predicate `bottomp` checks if b is the last element in st.

```
DEFINITION:
bottomp ($b$, $st$)
=   **if** $st \simeq$ **nil then f**
    **elseif** cdr ($st$) $\simeq$ **nil then** $b =$ car ($st$)
    **else** bottomp ($b$, cdr ($st$)) **endif**
```

The function `find-stack-of-block` returns the stack to which a given block belongs. If there is no such stack it returns **f**. Thus, this predicate can also be used to assert the existence of blocks in a state.

```
DEFINITION:
find-stack-of-block ($b$, $s$)
=   **if** $s \simeq$ **nil then f**
    **elseif** $b \in$ car ($s$) **then** car ($s$)
    **else** find-stack-of-block ($b$, cdr ($s$)) **endif**
```

The following examples show how `find-stack-of-block` may be used. The first example shows that the function returns the stack in which block `'c` is present in STATE1 and the second example shows that block `'g` does not exist in STATE1.

THEOREM: find-stack-example1
find-stack-of-block (`'c`, STATE1) = `'(a b c)`

THEOREM: find-stack-example2
$\neg$ find-stack-of-block (`'g`, STATE1)

Now we can define `ontable` as follows.

Definition:
ontable $(b,\ s)$ = bottomp $(b$, find-stack-of-block $(b,\ s))$

A block *b1* is on top of another block *b2* in a state *s* if there is a stack in *s* in which *b1* and *b2* occur consecutively. We define `on` in terms of `consecp`, a predicate that checks if two blocks occur consecutively in a stack.

Definition:

```
consecp ($b1$, $b2$, $st$)
=  **if** $st \simeq$ **nil then f**
   **elseif** cdr ($st$) $\simeq$ **nil then f**
   **elseif** car ($st$) = $b1$ **then** $b2$ = cadr ($st$)
   **else** consecp ($b1$, $b2$, cdr ($st$)) **endif**
```

Definition:
on $(b1,\ b2,\ s)$ = consecp $(b1,\ b2$, find-stack-of-block $(b1,\ s))$

A block is clear in a state if there is a stack in the state of which it is the topmost block.

Definition: clear $(b,\ s)$ = assoc $(b,\ s)$

Let us now define predicates on actions and plans. Actions are terms in the logic that are interpreted by *resultlist*. Below we give constructors and predicates for the move and unstack actions.

Definition: move $(b1,\ b2)$ = list (`'move`, $b1$, $b2$)

Definition: unstack (b) = list (`'unstack`, b)

Definition: movep (x) = (car (x) = `'move`)

Definition: unstackp (x) = (car (x) = `'unstack`)

Thus, `'(move a b)` is a term that stands for the action of moving block `'a` to the top of block `'b` and `'(unstack c)` is a term that stands for the action of unstacking block `'c`. The same action may be executed in many different states with different effects. For instance, `'(move a b)` is legal in a state where `'a` and `'b` are clear but is illegal otherwise.

The following predicate on plans is needed to express theorems that talk about existence of plans.

DEFINITION: actionp (x) = (movep (x) $\vee$ unstackp (x))

DEFINITION:
planp (x)
= **if** $x \simeq$ **nil then t**
  else actionp (car (x)) $\wedge$ planp (cdr (x)) **endif**

Here is a simple example of a plan.

DEFINITION:
PLAN1 = `'((unstack a) (move b a) (move c b))`

THEOREM: planp-example1
planp (PLAN1)

Having defined predicates on states, actions and plans we are ready to define the function `resultlist` which computes the state got as a result of executing a list of actions (plan) in a given state. *Resultlist* calls the function *result* which takes an action a and a state s as input and returns the state got as a result of executing a in s. *Result* simulates an action and computes the next state by calling Lisp programs `res-move` and `res-unstack` that specify the move and unstack actions respectively.

`Res-move` accepts as input blocks *b1* and *b2* and a state s. It checks if the preconditions for moving *b1* to the top of *b2* are satisfied in s and if so, calls `exec-move` to compute the new state. If not, it returns an error state, a list whose car is `'failed`. Notice that such an error state is distinct from all legal states since the car of every legal state is a stack and all stacks are non-empty lists. As preconditions, we require that *b1* be distinct from *b2* and that *b1* and *b2* be clear in s. The complete effects of a successful move action are specified using `exec-move`. `Exec-move` accepts *b1*, *b2* and s as input and returns the state got as a result of moving *b1* to the top of *b2* in s. All possible distinct circumstances in which the move action may be executed are taken care of by doing a case analysis on s. To move $b1$ to the top of $b2$ in state s, `exec-move` deletes the stacks containing $b1$ and $b2$ from s and then conses the stack got by adding *b1* to the top of *b2*'s stack. It returns the list of stacks l so obtained as the new state if *b1* is on the table in s. Otherwise the new state is got by consing the stack containing the rest of the blocks of $b1$'s stack onto l.

The function `delete` deletes the first occurrence of an element in a list.

DEFINITION:
delete (x, l)

```
= if $l \simeq$ nil then $l$
  elseif $x =$ car ($l$) then cdr ($l$)
  else cons (car ($l$), delete ($x$, cdr ($l$))) endif
```

DEFINITION:

```
exec-move ($b1$, $b2$, $s$)
= if cdr (assoc ($b1$, $s$)) $\simeq$ nil
  then cons (cons ($b1$, assoc ($b2$, $s$)),
             delete (assoc ($b1$, $s$), delete (assoc ($b2$, $s$), $s$)))
  else cons (cons ($b1$, assoc ($b2$, $s$)),
             cons (cdr (assoc ($b1$, $s$)),
                   delete (assoc ($b1$, $s$), delete (assoc ($b2$, $s$), $s$)))) endif
```

DEFINITION:

```
res-move ($b1$, $b2$, $s$)
= if ($b1 = b2$) $\lor$ ($\neg$ clear ($b1$, $s$)) $\lor$ ($\neg$ clear ($b2$, $s$))
  then list ('failed, 'res-move, $s$)
  else exec-move ($b1$, $b2$, $s$) endif
```

Similarly `res-unstack` returns an error state whose car is `'failed` if the preconditions for unstacking a block are not satisfied. Otherwise, it calls `exec-unstack` to carry out the effects of the unstack action.

DEFINITION:

```
exec-unstack ($b$, $s$)
= if cdr (assoc ($b$, $s$)) $\simeq$ nil then cons (list ($b$), delete (assoc ($b$, $s$), $s$))
  else cons (cdr (assoc ($b$, $s$)),
             cons (list ($b$), delete (assoc ($b$, $s$), $s$))) endif
```

DEFINITION:

```
res-unstack ($b$, $s$)
= if $\neg$ clear ($b$, $s$) then list ('failed, 'unstack, $s$)
  else exec-unstack ($b$, $s$) endif
```

The function `result` accepts an action a and a state s as input and behaves as follows. If s is an error state it is returned. Otherwise, the state got as a result of executing a in s is computed by calling `res-move` or `res-unstack`.

DEFINITION:

```
result ($a$, $s$)
= if car ($s$) = 'failed then $s$
  elseif movep ($a$) then res-move (cadr ($a$), caddr ($a$), $s$)
  else res-unstack (cadr ($a$), $s$) endif
```

`Resultlist` computes the state got as a result of executing a plan l in s.

DEFINITION:
resultlist (l, s)
= **if** $l \simeq$ **nil then** s
  else resultlist (cdr (l), result (car (l), s)) **endif**

Here are a few examples that show how `result` works. We use STATE1 and PLAN1 defined earlier. The first two examples show how legal move and unstack actions are executed and the next two show that error states are returned when the preconditions of actions are not met in the given state.

THEOREM: legal-move-example
result (move (`'a`, `'d`), STATE1) = `'((a d e) (b c))`

THEOREM: legal-unstack-example
result (unstack (`'a`), STATE1) = `'((b c) (a) (d e))`

THEOREM: illegal-move-example
result (move (`'b`, `'d`), STATE1)
= `'(failed res-move ((a b c) (d e)))`

THEOREM: illegal-unstack-example
result (unstack (`'e`), STATE1)
= `'(failed unstack ((a b c) (d e)))`

The following examples show how `resultlist` executes legal and illegal plans. Since the result of executing any action in an error state is the error state, it is impossible to achieve a legal state using illegal actions.

THEOREM: resultlist-legal
resultlist (PLAN1, STATE1) = `'((c b a) (d e))`

THEOREM: resultlist-illegal
resultlist (`'((unstack b) (move a c))`, STATE1)
= `'(failed unstack ((a b c) (d e)))`

2.2 Examples of Recursive Plans

We are now ready to demonstrate how plans to solve general problems can be expressed and verified using our formalization. Below we give a number of examples of plans with tests and recursion. We express such plans as plan generating Lisp programs that output a sequence of actions as a function of the input state and other parameters. The examples given below show how more than one plan to solve a problem can be expressed using our domain theory and how theorems used to verify a solution to one problem can be used for verifying a solution to another.

2.2.1 How to Clear a Block

Recall the problem of clearing a block mentioned in Chapter 1 in which the initial condition only specifies that there is a block b that exists in a state s. Our plan (given in [72]) to clear the block was to unstack repeatedly the blocks above it one by one until it became clear. We also showed that the same problem could also be posed as the following requirements specification for which we can use our plan as an initial program design or prototype.

> Write a PASCAL program that would take a blocks world state and a block as input and produce as output a blocks world state in which the given block is clear. The output state must have the same set of blocks as the input state.

To prove that the plan satisfies the above requirements specification, we must prove that the initial state and final state have the same set of blocks in addition to showing that the plan clears the block successfully.

The plan to clear a block may be expressed by the following plan generating program:

```
DEFINITION:
makeclear-gen ($b$, $s$)
= if ($\neg$ find-stack-of-block ($b$, $s$)) $\vee$ ($\neg$ bw-statep ($s$)) then f
  elseif $b$ = car (find-stack-of-block ($b$, $s$)) then nil
  else cons (unstack (car (find-stack-of-block ($b$, $s$))),
             makeclear-gen ($b$,
                            result (unstack (car (find-stack-of-block ($b$,
                                                                      $s$))),
                                    $s$))) endif
```

The program produces a sequence of actions that will clear a block b if it exists in the given state s. The program operates by finding the stack in s to which b belongs. If b is not at the top of its stack, then an action to clear the top of its stack is generated and consed on to the plan got by executing `makeclear-gen` recursively on the state got by simulating the action in s. To make the proof of termination of the above function relatively straightforward, we have the program return **f** (i.e. terminate) when its arguments are not legal. This is harmless since the program is intended to be used only when the given block b exists in the given blocks world state s. Viewed as a Lisp program, `makeclear-gen` is a simple, special-purpose plan generator that can be "run" on particular arguments to generate an appropriate sequence of actions. But, because of its status as a function in the logic, `makeclear-gen` may also be used to verify mechanically the plan to clear a block. The following correctness theorem about `makeclear-gen` states that the plans generated by it can be used to solve the problem of clearing a block.

THEOREM: makeclear-works1
(bw-statep (s)
$\wedge$ find-stack-of-block (b, s)
$\wedge$ ($p1$ = makeclear-gen (b, s))
$\wedge$ ($s1$ = resultlist ($p1$, s)))
$\rightarrow$ (bw-statep ($s1$) $\wedge$ planp ($p1$) $\wedge$ clear (b, $s1$))

The above theorem may be read as follows: If b is a block in state s then makeclear (b, s) is a plan that if executed in s results in a legal state $s1$ in which b is clear. The theorem itself can be "tested" on concrete data structures. Thus, we can test that `makeclear-gen` can clear block `'c` in state STATE1 by proving the following theorem.

THEOREM: makeclear-gen-ex1
((p = makeclear-gen (`'c`, STATE1)) $\wedge$ (s = resultlist (p, STATE1)))
$\rightarrow$ (bw-statep (s) $\wedge$ planp (p) $\wedge$ clear (`'c`, s))

Theorems involving completely specified data structures are proved *automatically* by the theorem prover. The ability to "test" plans or program designs in specific situations is particularly useful for rapid prototyping because prototypes can be subject to *acceptance tests* [64] using particular scenarios that a customer is familiar with. This removes errors early in the software life cycle and reduces costs.

We will now explain briefly how the theorem prover was guided interactively to the proof of the theorem make-clear-works1. Recall that, among functions not defined using `eval$`, only terminating functions are admitted as definitions in the logic. The metric for termination of `makeclear-gen` is the number of blocks in the stack containing b in s. It is clear that the metric decreases and is bounded below since the topmost block from the stack containing b is removed with each recursive call. The theorem prover cannot "invent" this metric; so it was supplied by the following definition.

DEFINITION: m3 (b, s) = len (find-stack-of-block (b, s))

There are three things we must prove about `makeclear-gen`: that it generates valid plans, that the plans generated return a legal blocks world state when executed in an initial state, and that the given block is clear in the final state. Proving that `makeclear-gen` generates valid plans is done easily by induction and proving that the plans achieve the goal is also relatively straightforward once `makeclear-gen` is itself admitted as a definition. A major portion of theorem proving effort goes into showing that the final states got by executing the generated plans are legal blocks world states. From the definition of `bw-statep`, it is evident that we must prove that the final state got by executing a generated plan is a set of stacks and that the stacks are pairwise disjoint. Since `makeclear-gen` only generates unstack actions, we need the following lemma that says that unstacking a clear block in a blocks world state results in a blocks world state.

THEOREM: unstack-sit1
(bw-statep (s) $\wedge$ clear (b, s)) $\rightarrow$ bw-statep (result (unstack (b), s))

Some of the lemmas we need for proving the above theorem are the following. Since the effect of unstacking a block is described by deleting and adding stacks to the input state, we must show that deleting stacks from a state results in a legal blocks world state, that the new lists of blocks to be added are valid stacks, and that they are disjoint from the remaining stacks. The following theorem says that deleting an element in a blocks world state results in a blocks world state.

THEOREM: del-set1
bw-statep (s) $\rightarrow$ bw-statep (delete (x, s))

The new stacks added are the singleton list with the block b that was unstacked and the stack containing the rest of blocks in b's original stack (if any). The following lemmas were proved to show that the new stacks are disjoint from the remaining stacks in the original state.

THEOREM: unstackl2
(bw-statep (s) $\wedge$ clear (b, s) $\wedge$ listp (cdr (assoc (b, s))))
$\rightarrow$ disjointlist (cdr (assoc (b, s)), delete (assoc (b, s), s))

THEOREM: disjointlist-single
(bw-statep (s) $\wedge$ clear (b, s))
$\rightarrow$ disjointlist (list (b), delete (assoc (b, s), s))

To prove these, we require a number of facts about the functions `disjoint` and `disjointlist` as shown in Section A.1 of Appendix A.

For proving that the plan meets the requirements specification, we must prove in addition that the set of blocks in the final state and the set of blocks in the initial state are equal. To state that two states have the same set of blocks we need the following two functions. `Set-equal` ascertains that the two given lists contain the same elements by deleting the members one by one while `set-of-blocks` returns the set of blocks in a state by appending all the stacks together.

```
DEFINITION:
set-equal ($s1$, $s2$)
=  if $s1 \simeq$ nil then $s2 \simeq$ nil
   else (car ($s1$) $\in s2$)
        $\wedge$ set-equal (cdr ($s1$), delete (car ($s1$), $s2$)) endif
```

```
DEFINITION:
set-of-blocks ($s$)
=  if $s \simeq$ nil then nil
   else append (car ($s$), set-of-blocks (cdr ($s$))) endif
```

We can now state that all the plans generated by `makeclear-gen` preserve the set of blocks in the initial state. †

```
THEOREM: makclear-gen-preserves-blocks
 (bw-statep ($s0$)
  $\wedge$ find-stack-of-block ($b$, $s0$)
  $\wedge$ ($p1$ = makeclear-gen ($b$, $s0$))
  $\wedge$ ($s1$ = resultlist ($p1$, $s0$)))
 $\rightarrow$ set-equal (set-of-blocks ($s0$), set-of-blocks ($s1$))
```

2.2.2 Other Examples

We would like to show that our specification of the blocks world is adequate for expressing other general problems and for verifying other solutions to the problem of clearing a block. Here are some more examples of programming problems and solutions in the blocks world. All the solutions are, as before, expressed as plan generating Lisp programs. The proofs of these theorems clearly demonstrate how a single domain theory may be shared across many problems of a domain.

Consider a plan to invert a tower on top of another. Once again this requires testing and recursion: move one block after another from one tower to the top of the other until there are no blocks to be transferred. `Invert-gen` is a program to generate the required sequence of actions in various states.

```
DEFINITION:
invert-gen ($st1$, $st2$, $s$)
=  if $st1 \simeq$ nil then nil
   else cons (move (car ($st1$), car ($st2$)),
              invert-gen (cdr ($st1$),
                          cons (car ($st1$), $st2$),
                          result (move (car ($st1$), car ($st2$)), $s$))) endif
```

To state that `invert-gen` generates correct plans for inverting one stack *st1* on top of another stack *st2*, we must say that the tower got by appending the reverse of *st1* to *st2* belongs to the final state. Here is a function to reverse a list followed by the statement of the theorem that asserts that `invert-gen` works.

```
DEFINITION:
reverse ($l$)
=  if $l \simeq$ nil then nil
   else append (reverse (cdr ($l$)), list (car ($l$))) endif
```

```
THEOREM: invert-works2
 (bw-statep ($s$)
  $\land$  ($st1 \in s$)
  $\land$  ($st2 \in s$)
  $\land$  ($st1 \neq st2$)
  $\land$  ($p1$ = invert-gen ($st1$, $st2$, $s$))
  $\land$  ($s1$ = resultlist ($p1$, $s$)))
 $\rightarrow$  (bw-statep ($s1$)
      $\land$  planp ($p1$)
      $\land$  (append (reverse ($st1$), $st2$) $\in s1$))
```

Here is an example to test the above theorem.

```
THEOREM: invert-gen-ex1
 (($p1$ = invert-gen ('(a b c), '(d e), STATE1))
  $\land$  ($s1$ = resultlist ($p1$, STATE1)))
 $\rightarrow$  (bw-statep ($s1$) $\land$ planp ($p1$) $\land$ ('(c b a d e) $\in s1$))
```

Below we give two more examples: a plan to put all the blocks in a given stack on the table and a plan to put all the blocks in all the stacks on the table. Notice that both of these plans could also be used for clearing a block although they are less efficient in the number of actions needed than the original plan. Similarly, the plan to put the blocks in all the towers on the table can be used to put the blocks in a particular tower on the table.

The following program `unstack-tower-gen` generates plans to put all the blocks in a stack on the table. It generates the action to unstack the topmost block in the given stack *st* and recursively calls itself with the remaining stack and the state got as a result of unstacking the topmost block of *st*.

```
DEFINITION:
unstack-tower-gen ($st$, $s$)
=  if $st \simeq$ nil then nil
   elseif cdr ($st$) $\simeq$ nil then nil
   else cons (unstack (car ($st$)),
              unstack-tower-gen (cdr ($st$),
                                 result (unstack (car ($st$)), $s$))) endif
```

To state that all blocks in a stack *st* are on the table in a state *s*, we introduce the following predicate.

```
DEFINITION:
ontable-list ($st$, $s$)
=  if $st \simeq$ nil then t
   else (find-stack-of-block (car ($st$), $s$) = list (car ($st$)))
        $\land$ ontable-list (cdr ($st$), $s$) endif
```

That `unstack-tower-gen` accomplishes the desired goal is stated as follows.

```
THEOREM: unstack-tower-works1
 (bw-statep ($s$)
  $\land$ ($st \in s$)
  $\land$ ($p1$ = unstack-tower-gen ($st$, $s$))
  $\land$ ($s1$ = resultlist ($p1$, $s$)))
 $\rightarrow$ (bw-statep ($s1$) $\land$ planp ($p1$) $\land$ ontable-list ($st$, $s1$))
```

The plan to put all the blocks of all the towers on the table is carried out by a program that unstacks repeatedly a block not on the table in the given state (returned by the function `get-block`) until all the blocks are on the table. The function `get-block` goes through a blocks world state and returns a block that is not on the table. If all the blocks are on the table, `get-block` returns **f**.

```
DEFINITION:
get-block ($s$)
=  if $s \simeq$ nil then f
   elseif $\neg$ bw-statep ($s$) then f
   elseif cdar ($s$) $\simeq$ nil then get-block (cdr ($s$))
   else caar ($s$) endif
```

Given below are `unstack-all-towers-gen`, a program that generates plans to unstack all the blocks in a state, and a formal statement of its correctness.

```
DEFINITION:
unstack-all-towers-gen ($s$)
= if get-block ($s$)
  then cons (unstack (get-block ($s$)),
             unstack-all-towers-gen (result (unstack (get-block ($s$)), $s$)))
  else nil endif
```

THEOREM: unstack-all-towers-works
(bw-statep (s)
 $\wedge$ ($p1$ = unstack-all-towers-gen (s))
 $\wedge$ ($s1$ = resultlist ($p1$, s)))
$\rightarrow$ (bw-statep ($s1$) $\wedge$ planp ($p1$) $\wedge$ ($\neg$ get-block ($s1$)))

We give some more examples to demonstrate the expressiveness of our plan representation. However, the rest of the general theorems in this section have not been checked by the prover. Our plan representation makes it easy to use concepts such as numbers that may be needed for plan generation. Consider the problem of building a tower of n blocks given that there are at least n stacks in the initial state. One plan to do that is to unstack the topmost block of one of the towers and then move one by one the topmost blocks from $n-1$ other towers to the top of the first block. To keep track of the number of blocks moved, we need arithmetic. The following is a program to generate the above plan.

```
DEFINITION:
build-towern ($s$, $n$)
= if $n \simeq 0$ then nil
  elseif $n = 1$ then list (unstack (caar ($s$)))
  else append (build-towern (cdr ($s$), $n - 1$),
               list (move (caar ($s$), caadr ($s$)))) endif
```

Here is an example of a plan generated by `build-towern`.

THEOREM: build-towern-ex1
build-towern (STATE1, 2) = `'((unstack d) (move a d))`

To state the correctness of `build-towern`, we need a predicate that checks if there is a tower of height n in a state.

```
DEFINITION:
exists-towern (n, s)
=  if s ≃ nil then f
   elseif len (car (s)) = n then t
   else exists-towern (n, cdr (s)) endif
```

The correctness theorem of `build-towern` and a concrete example of how it works are given below. †

```
THEOREM: build-towern-works
 (bw-statep (s)
  ∧  (len (s) > n)
  ∧  (n ≄ 0)
  ∧  (p1 = build-towern (s, n))
  ∧  (s1 = resultlist (p1, s)))
  →  (bw-statep (s1) ∧ planp (p1) ∧ exists-towern (n, s1))
```

```
THEOREM: build-towern-ex2
 exists-towern (2, resultlist (build-towern (STATE1, 2), STATE1))
```

Here is a strategy that can tranform any blocks world state *s1* to any other blocks world state *s2* with the identical collection of blocks as *s1*. To change *s1* to *s2*, simply put all the blocks in *s1* on the table and then build the stacks in *s2* one by one. If we do not have any efficiency restrictions, then we can use this strategy to solve any problem in the blocks world. Below we state the correctness of a program that generates actions using the above strategy.

We have already written and proved correct a program `unstack-all-towers-gen` that generates plans to put all the blocks that exist in a state on the table. We will write a program `form-state` to generate plans to construct all the stacks in a state *s2* when executed in a state in which all the blocks in *s2* are on the table.

The following auxiliary function `form-tower` generates plans to build a stack *st* using all the blocks that are on the table.

```
DEFINITION:
form-tower (st)
=  if st ≃ nil then nil
   elseif cdr (st) ≃ nil
   then append (form-tower (cdr (st)), list (unstack (car (st))))
   else append (form-tower (cdr (st)),
                list (move (car (st), cadr (st)))) endif
```

The function `form-state` forms all the stacks in a state by building one tower after another using form-tower.

DEFINITION:
form-state (*s2*)
= **if** *s2* $\simeq$ **nil then nil**
  else append (form-tower (car (*s2*)), form-state (cdr (*s2*))) **endif**

Now, the program to transform any initial state *s1* to any goal state *s2* with the same set of blocks can be written as follows.

DEFINITION:
transform (*s1*, *s2*) = append (unstack-all-towers1 (*s1*), form-state (*s2*))

The correctness theorem for the above plan generator can be stated as follows: If there are two states *s1* and *s2* with the same set of blocks then executing the plan generated by transform (*s1*, *s2*) in *s1* results in *s2*. †

THEOREM: planner-works
(bw-statep (*s1*)
 $\wedge$ bw-statep (*s2*)
 $\wedge$ set-equal (set-of-blocks (*s1*), set-of-blocks (*s2*)))
 $\rightarrow$ (resultlist (transform (*s1*, *s2*), *s1*) = *s2*)

2.3 Specification of Actions with Side-effects

To further clarify how our method can be used to specify domains involving actions that produce side-effects, we will specify a variation of the blocks world given in [28]. The set of states is the same as before. We are allowed only one action, move a block to the top of another block. This action has the side-effect of moving all the blocks above the block being moved along with it to the new location. Thus, we are allowed to move any block in this world even if they have blocks above them, but when a block is moved, all the blocks over it move along with it. Here is a simple formalization of this domain and a plan to form a single tower using all the existing blocks in a state. The formalization also demonstrates how theories can be shared across domains when small modifications to domain specifications are made. The definition of the predicate on blocks world states and the theorems about it can be used without change. Observe that we could also define the new move action as a plan that is realized in terms of the move and unstack actions of the previous blocks world. But since we want to demonstrate how atomic actions with side-effects can be specified, we will model the new move action as a separate atomic action.

We will use the same predicate `bw-statep` on the set of states. Let us call our new action, `move-over`. To move over a block x to the top of a block y, we have the following preconditions: x and y should be on distinct stacks (and must therefore be distinct) and y must be clear. The effect of the action is to move the stack of blocks in the stack containing x, from the topmost block upto (and including) x, to the top of y. The following function returns the list of blocks above a block b in a stack st.

```
DEFINITION:
stack-above ($b$, $st$)
= **if** $st \simeq$ **nil then nil**
  **elseif** $b$ = car ($st$) **then** list (car ($st$))
  **else** cons (car ($st$), stack-above ($b$, cdr ($st$))) **endif**
```

The following function finds the stack containing a given block x in a state s and uses `stack-above` to return the stack of blocks above it.

DEFINITION:
stack-above1 (x, s) = stack-above (x, find-stack-of-block (x, s))

Using these functions, we can define the relation `over` as follows.

DEFINITION:
over (x, y, s) = (($x \neq y$) $\land$ ($x \in$ stack-above1 (y, s)))

Thus, x is over y in s if it is one of the blocks in the stack above y in s. We implement the effects of moving x over y in state s as follows: we delete the stacks containing x and y from s and add two new stacks, one formed by appending the stack above x to the stack containing y in s and the other containing the rest (if any) of blocks in x's stack. The function `stack-below` returns the blocks below x in a stack st.

```
DEFINITION:
stack-below ($x$, $st$)
= **if** $st \simeq$ **nil then f**
  **elseif** $x$ = car ($st$) **then** cdr ($st$)
  **else** stack-below ($x$, cdr ($st$)) **endif**
```

The function `stack-below1` returns the stack below x in state s.

DEFINITION:
stack-below1 (x, s) = stack-below (x, find-stack-of-block (x, s))

`Exec-move-over` implements the effects of moving over x to the top of y in s. This function is called by `res-move-over` only when the preconditions of the action are satisfied in s. `Exec-move-over` forms the new set of stacks depending on whether there are blocks below x in the stack containing x in s.

```
DEFINITION:
exec-move-over ($x$, $y$, $s$)
=  if stack-below1 ($x$, $s$) $\simeq$ nil
   then cons (append (stack-above1 ($x$, $s$), assoc ($y$, $s$)),
              delete (find-stack-of-block ($x$, $s$), delete (assoc ($y$, $s$), $s$)))
   else cons (stack-below1 ($x$, $s$),
             cons (append (stack-above1 ($x$, $s$), assoc ($y$, $s$)),
                   delete (find-stack-of-block ($x$, $s$),
                           delete (assoc ($y$, $s$), $s$)))) endif
```

`Res-move-over` returns an error state whose car is `'failed` if the preconditions of the action are not satisfied.

```
DEFINITION:
res-move-over ($x$, $y$, $s$)
=  if ($\neg$ find-stack-of-block ($x$, $s$))
      $\vee$ (find-stack-of-block ($x$, $s$) = assoc ($y$, $s$))
      $\vee$ ($\neg$ clear ($y$, $s$))
   then list ('failed, 'move-over, $x$, $y$, $s$)
   else exec-move-over ($x$, $y$, $s$) endif
```

Below are a constructor for the action `move-over` and a predicate for the action `move-overp` followed by a predicate on plans that contain only the move-over action.

DEFINITION: move-over (x, y) = list (`'move-over`, x, y)

DEFINITION: move-overp (x) = (car (x) = `'move-over`)

```
DEFINITION:
planp-new ($x$)
=  if $x \simeq$ nil then t
   else move-overp (car ($x$)) $\wedge$ planp-new (cdr ($x$)) endif
```

We have written these definitions so that they can be directly added to our previous blocks world formalization. Thus, we call the *result* and *resultlist* functions for this blocks world as `result-new` and `resultlist-new`.

```
DEFINITION:
result-new (a, s)
=  if car (s) = 'failed then s
   else res-move-over (cadr (a), caddr (a), s) endif
```

```
DEFINITION:
resultlist-new (l, s)
=  if l ≃ nil then s
   else resultlist-new (cdr (l), result-new (car (l), s)) endif
```

Here are some examples that test our formalization.

```
THEOREM: new-ex1
 result-new (move-over ('b, 'd), '((a b c) (d e)))
 =  '((c) (a b d e))
```

```
THEOREM: new-ex2
 result-new (move-over ('b, 'e), '((a b c) (d e)))
 =  '(failed move-over b e ((a b c) (d e)))
```

```
THEOREM: new-ex3
 result-new (move-over ('c, 'd), '((a b c) (d e)))
 =  '((a b c d e))
```

```
THEOREM: new-ex4
 resultlist-new (list (move-over ('b, 'd), move-over ('e, 'c)),
                 '((a b c) (d e)))
 =  '((a b d e c))
```

The plan generating program to form a single tower out of all the blocks in a state s in which at least one stack exists is given below. This is followed by a statement of its correctness. The function `last` returns the last element of a list.

```
DEFINITION:
last (l)
=  if l ≃ nil then f
   elseif cdr (l) ≃ nil then car (l)
   else last (cdr (l)) endif
```

DEFINITION:
form-single-tower-gen (s)

```
= if (¬ bw-statep (s)) ∨ (s ≃ nil) then nil
  elseif cdr (s) ≃ nil then nil
  else cons (move-over (last (car (s)), caadr (s)),
             form-single-tower-gen (result-new (move-over (last (car (s)),
                                                          caadr (s)),
                                               s))) endif
```

THEOREM: move-over-plan-works
(bw-statep (s)
$\wedge$ listp (s)
$\wedge$ ($p1$ = form-single-tower-gen (s))
$\wedge$ ($s1$ = resultlist-new ($p1$, s)))
$\rightarrow$ (bw-statep ($s1$) $\wedge$ planp-new ($p1$) $\wedge$ (cdr ($s1$) $\simeq$ **nil**))

The list of events to check the above theorem is given in Section A.2 of Appendix A. The mechanical proof of the above theorem was relatively easy because the necessary theorems about functions such as `disjoint` and `disjointlist` used in the definition of the predicate on blocks world states, `bw-statep`, could be reused from Section A.1. This is the reason why Section A.2 which contains all the new events for formalizing this blocks world and verifying the plan to form a single tower is relatively short. The list of events in Section A.2 was loaded into the theorem prover after loading the sequence of events in Appendix A upto the event named find-stack-listp in Section A.1. (The rest of events in Section A.1. were not needed for the proof). Thus, theorems can be shared across domains involving similar concepts when a class of domains is mechanized in the same logic.

2.4 Conclusion

In this chapter, we used our method of specifying problem domains to specify some variations of the blocks world. Plans for solving general problems in the blocks world were specified as plan generating Lisp programs and verified by proving the plan generating programs correct. We also saw how actions with side-effects can be specified and how theorems can be shared across problems in the same domain and across similar problem domains. As mentioned in Chapter 1, this representation of plans does not allow us to use plans as atomic (complex) actions in other plans. This defect is remedied in Chapter 5.

Chapter 3

The Mutilated Checkerboard Problem

The bulk of this chapter is devoted to a specification of the $n \times n$ mutilated checkerboard problem and an interactive mechanical proof, using colors, of the impossibility of covering a mutilated $n \times n$ checkerboard completely with dominoes. As before, we are interested in specifying the problem domain so that all possible solutions can be expressed and verified. We regard the $n \times n$ mutilated checkerboard problem as one of the class of general problems that can be posed in the problem domain defined by two dimensional space and domino placement actions. We formalize the well-known parity argument that requires the use of colors of squares as a proof by mathematical induction on the length of plans executed starting with an empty board. We have chosen this example for three reasons. First, proofs that show that a plan or a class of plans does not satisfy a requirements specification seem to be needed during software development [56]. Therefore, a mechanical system for program development must be able to carry out such proofs even when they involve concepts not directly mentioned in the problem statement. Second, the problem domain—two-dimensional space and actions that change properties of points in space—is realistic and of interest in both AI applications such as robot motion planning and programming applications such as graphics. Finally, although the proof has been repeatedly posed as a challenge for *automatic discovery* by machines [98, 75, 78, 107], McCarthy [83] informs us that he knows of “no work on making a computer do it, interactively or otherwise”.

Apart from the proof, some plans for solving general problems in this domain are also given along with theorems that must be proved to verify them. The sequence of events to the Boyer-Moore theorem prover for formalizing the checkerboard problem domain and proving the impossibility of covering a mutilated checkerboard is included in Appendix B.

3.1 The Mutilated Checkerboard Problem

The mutilated checkerboard problem has been used frequently [98, 75, 78, 107] to demonstrate the limitations of representations used by problem solving programs and theorem provers. The problem is usually stated for an ordinary 8×8 checkerboard as follows [107, 121]:

> An ordinary chess board has had two squares—one at each end of a diagonal—removed. There is on hand an unlimited supply of dominoes, each of which is large enough to cover exactly two adjacent squares of the board. Is it possible to lay the dominoes on the mutilated chess board in such a manner as to cover it completely?

In the more difficult version of the problem, the squares on the board are assumed to be indistinguishable by color.

A solution [107] to the problem which researchers have wanted problem solving programs to discover is the following convincing argument that such a tiling is impossible.

> Let us color the squares alternately white and black (as on the usual chess board). The two missing squares have the same color. Thus, the mutilated board has an unequal number of white and black squares. Since each domino covers exactly one black square and one white square, any covering by dominoes covers an equal number of white and black squares and so the desired covering does not exist.

It is fairly clear that the problem can be stated for any $n \times n$ board for $n > 1$ and that the simple argument solves the problem in every case. Of course, the problem is easy if the board has an odd number of rows and columns because both the actual board and the corresponding mutilated one would then have an odd number of squares. Since placing dominoes on a board results in an even number of squares being covered, neither a board with an odd number of rows and columns nor the corresponding mutilated one can be covered completely. However, the argument using colors holds even in the case when the board has an odd number of rows and columns because the number of squares of one color in such a board is one greater than the number of squares of the other color. Thus, removing two squares of the same color would still make the number of white and black squares on the mutilated board unequal.

Newell [98] observed that the argument can be formalized as a proof by induction on the number of domino placement actions done starting with an empty board assuming that the colors of the squares on the board are given. Using mathematical induction, we can exhaustively "test" the infinite number of possible sequences of actions and show that none of them can be used to cover a mutilated $n \times n$ board.

One previous attempt to mechanize the checkerboard solution was given by Wos et al. [121]. However, their effort differs from what we want in two respects. First, they propose a solution to the 8×8 checkerboard problem and not the $n \times n$ problem as in our case. Secondly, they reduce the state space of the problem by considering a particular order for tiling the board and provide clauses for searching exhaustively this reduced search space by theorem proving. Their proof also does not involve the use of colors. On the other hand, our goal is to obtain a specification of the problem domain so that various solutions to this problem and other problems that arise in this context can be expressed and verified interactively.

3.2 Formalization of the Mutilated Checkerboard Problem

The theorem we would like to express may be informally stated as follows:

There does not exist a sequence of legal domino placement actions that when executed in a state in which none of the squares of an $n \times n$ board, $n > 1$, are covered will result in a state in which the board is completely covered.

Our method of formalizing problem domains requires us to define a predicate on states, a predicate on actions, the state transition function *result* that maps an action a and a state s to the state got by executing a in s and the interpreter *resultlist* that takes as input a sequence of actions l and a state s and returns the state got by executing l in s. For this we must first choose appropriate data structures to represent states, including error states, and actions. The states of this problem domain are partially covered boards of all dimensions, i.e., two dimensional space in which points may or may not have the property of being covered by a domino and the actions are the domino placement actions. Once again, there are several ways to represent the state of a checkerboard. If we "think" Lisp instead of first-order term representation as in Prolog, a representation of checkerboard states that suggests itself is a two-dimensional array of squares with a token `'N` or `'C` in each position of the array to indicate whether the square in that position is covered or not. The representation of checkerboard states that we have chosen is simpler. We represent the states by the list of squares that are currently covered with the squares that do not belong to the list being those that are not covered. We label the rows and columns of a checkerboard from 0 to n; so a checkerboard has an even number of rows if n is odd and vice versa. The square at row numbered x and column numbered y is represented as cons$(x,\ y)$, i.e., by its coordinates. With this representation we merely have to test for membership in a list to check for coveredness of a square in a state. Notice that this representation of states generalizes the problem domain to include states in which solitary squares may be covered, i.e., states other than those in which all covered squares are covered by dominoes. This generalization is harmless and actually helps simplify the representation.

In the following presentation, we will take n to be the number of the last row (or column) of a board. As we introduce definitions of Lisp functions, we will prove theorems that show the result of executing the functions on ground arguments. Below are predicates `squarep` and `square-listp` on squares and lists of squares respectively.

DEFINITION:
squarep (x) = (listp (x) $\wedge$ (car (x) $\in$ **N**) $\wedge$ (cdr (x) $\in$ **N**))

DEFINITION:
square-listp (x)
= **if** $x \simeq$ **nil then t**
  else squarep (car (x)) $\wedge$ square-listp (cdr (x)) **endif**

Board states are defined as lists of squares.

DEFINITION: board-statep (x) = square-listp (x)

Here are some examples of actual board states and some "executions" of the predicate `board-statep`.

DEFINITION:
STATE1 = `'((3 . 4) (3 . 5) (4 . 6) (4 . 7))`

THEOREM: board-state-example1
board-statep (STATE1)

THEOREM: board-state-example2
board-statep (**nil**)

Having represented states, let us turn to actions. For this, we must represent dominoes. A domino is represented by a pair of adjacent squares that fall within a board. The predicate `squarenp` checks if a given square falls within a board whose rows and columns are numbered from 0 through n.

DEFINITION:
squarenp (x, n) = (squarep (x) $\wedge$ (car (x) $\leq n$) $\wedge$ (cdr (x) $\leq n$))

If we take adjacency between the two squares representing a domino to be the usual symmetric relation, then we end up with two dominoes for every pair of adjacent squares depending on the ordering of the squares. Considering both orientations of the same domino would unnecessarily complicate reasoning about the effects of all possible domino placement actions by adding an extra case. Without loss of generality, we restrict the car of a domino to be closer to square `'(0 . 0)` than its cdr and define the adjacency predicate `adjp` accordingly. That is, cons (`'(0 . 1)`, `'(0 . 2)`) is a domino in a board numbered from 0 through 2 or greater whereas cons (`'(0 . 2)`, `'(0 . 1)`) is not.

DEFINITION:
adjp ($s1$, $s2$)
$= \quad (((\mathrm{car}(s1) = \mathrm{car}(s2)) \land ((1 + \mathrm{cdr}(s1)) = \mathrm{cdr}(s2)))$
$\quad \lor \quad ((\mathrm{cdr}(s1) = \mathrm{cdr}(s2)) \land ((1 + \mathrm{car}(s1)) = \mathrm{car}(s2))))$

DEFINITION:
dominop (x, n)
$= \quad (\mathrm{squarenp}(\mathrm{car}(x), n)$
$\quad \land \quad \mathrm{squarenp}(\mathrm{cdr}(x), n)$
$\quad \land \quad \mathrm{adjp}(\mathrm{car}(x), \mathrm{cdr}(x)))$

DEFINITION: DOMINO1 = `'((0 . 1) 0 . 2)`

DEFINITION: ILLEGAL-DOMINO = `'((0 . 2) 0 . 1)`

THEOREM: adjacency-example1
adjp (car (DOMINO1), cdr (DOMINO1))

THEOREM: adjacency-example2
¬ adjp (car (ILLEGAL-DOMINO), cdr (ILLEGAL-DOMINO))

THEOREM: dominop-example1
¬ dominop (ILLEGAL-DOMINO, `3`)

THEOREM: dominop-example2
¬ dominop (DOMINO1, `1`)

THEOREM: dominop-example3
dominop (DOMINO1, `2`)

We model the action of placing a domino on a board whose rows and columns are numbered from 0 through n by a list whose second element is a domino.

DEFINITION: placep (x, n) = dominop (cadr (x), n)

Here is a constructor for actions that uses the litatom `'place` as the car of actions for readability.

DEFINITION: place (x) = list (`'place`, x)

We are ready to define the transition function *result*. In formalizing the effects of domino placement actions, we must take into account the preconditions that disallow us from placing a domino on a square that is already covered. We ensure that illegal actions are disallowed by making the function *result* return "error states" of the form list (`'failed`, . . .) whenever preconditions of an action are not satisfied in a state. Notice that error states are distinct from board states: there is no board state whose car is `'failed` since `'failed` is not a pair. We also ensure that the result of doing an action in an error state is an error state. This makes it impossible to bring about a legal board state by doing illegal actions or any mixture of legal and illegal actions.

The function `res-place` checks the preconditions for placing a domino on a board and simulates the effects of placing a domino by adding the squares covered by the domino to the given board state. The function `result` ensures that the result of doing an action in an error state is the error state.

DEFINITION:
res-place (x, s)
= **if** (car (x) $\in$ s) $\vee$ (cdr (x) $\in$ s)
  then list (`'failed`, `'place`, x, s)
  else cons (car (x), cons (cdr (x), s)) **endif**

DEFINITION:
result (a, s)
= **if** car (s) = `'failed` **then** s
  else res-place (cadr (a), s) **endif**

The following examples show how *result* adds the squares on which a domino is placed to a board state and how it disallows placement of a domino on squares already covered by returning an error state. Placing DOMINO1 is a legal action on an 8×8 board but an illegal action on a 2×2 board, i.e., a board in which the rows and columns are numbered from 0 through 1.

DEFINITION: ACTION1 = place (DOMINO1)

THEOREM: placep-example1
placep (ACTION1, 7)

THEOREM: placep-example2
$\neg$ placep (ACTION1, 1)

The following examples show that `result` disallows placement of a domino on top of an existing domino.

THEOREM: result-example1
result (ACTION1, STATE1)
```
=  '((0 . 1)
     (0 . 2)
     (3 . 4)
     (3 . 5)
     (4 . 6)
     (4 . 7))
```

DEFINITION: STATE2 = result (ACTION1, STATE1)

THEOREM: error-state-example1
result (ACTION1, STATE2)
```
=  '(failed place
     ((0 . 1) 0 . 2)
     ((0 . 1)
      (0 . 2)
      (3 . 4)
      (3 . 5)
      (4 . 6)
      (4 . 7)))
```

Execution of an action in an error state results in the same error state.

THEOREM: error-state-example2
result (ACTION1, result (ACTION1, STATE2))
```
=  '(failed place
     ((0 . 1) 0 . 2)
     ((0 . 1)
      (0 . 2)
      (3 . 4)
      (3 . 5)
      (4 . 6)
      (4 . 7)))
```

Plans are represented as lists of actions. The Lisp program `place-planp` recognizes those plans that can be executed on a board whose last row/column has the label n. The function `resultlist` returns the state got by executing a plan in a given state.

DEFINITION:
place-planp (x, n)
= **if** $x \simeq$ **nil then t**
else placep (car (x), n) $\wedge$ place-planp (cdr (x), n) **endif**

DEFINITION:
resultlist (l, s)
= **if** $l \simeq$ **nil then** s
  else resultlist (cdr (l), result (car (l), s)) **endif**

Here are some examples of plans and how they change the state of the board. Notice that for a plan to be successful every one of the actions in the sequence must result in a legal state when executed.

DEFINITION: BAD-PLAN = list (ACTION1, ACTION1)

DEFINITION: ACTION2 = place (`'((4 . 1) 4 . 2)`)

DEFINITION: GOOD-PLAN = list (ACTION1, ACTION2)

THEOREM: plan-ex1
place-planp (GOOD-PLAN, 7)

THEOREM: plan-ex2
place-planp (BAD-PLAN, 7)

THEOREM: plan-ex3
resultlist (GOOD-PLAN, STATE1)

```
=  '((4 . 1)
     (4 . 2)
     (0 . 1)
     (0 . 2)
     (3 . 4)
     (3 . 5)
     (4 . 6)
     (4 . 7))
```

THEOREM: plan-ex4
resultlist (BAD-PLAN, STATE1)

```
=  '(failed place
     ((0 . 1) 0 . 2)
     ((0 . 1)
      (0 . 2)
      (3 . 4)
      (3 . 5)
      (4 . 6)
      (4 . 7)))
```

This completes the representation of the problem domain. To express the main theorem, we need predicates to recognize initial states and goal states. Empty boards of all dimensions are represented as **nil**. We define the predicate `all-covered-except-cornerp` to recognize goal states by checking if all squares of a mutilated board numbered from 0 through n are covered in a given state. We generate all squares of the mutilated board and then define `all-covered-except-cornerp` to check for set equality with the generated set of squares. The function `make-row` returns a list of all squares in row m from column 0 through column n.

```
DEFINITION:
make-row (m, n)
=  if n ≃ 0 then list (cons (m, 0))
   else append (make-row (m, n − 1), list (cons (m, n))) endif
```

The list of squares in the sixth row, i.e., the row numbered 5, of an 8×8 board is given by the following theorem.

```
THEOREM: row5
 make-row (5, 7)
 =  '((5 . 0)
      (5 . 1)
      (5 . 2)
      (5 . 3)
      (5 . 4)
      (5 . 5)
      (5 . 6)
      (5 . 7))
```

The function `make-all-rows` returns a list of all squares in a board whose rows and columns are numbered from 0 through m and 0 through n respectively.

```
DEFINITION:
make-all-rows (m, n)
=  if m ≃ 0 then make-row (0, n)
   else append (make-all-rows (m − 1, n), make-row (m, n)) endif
```

Here is a list of squares of a 2×2 board.

```
THEOREM: twobytwo
 make-all-rows (1, 1) = '((0 . 0) (0 . 1) (1 . 0) (1 . 1))
```

To get the set of squares of a mutilated board numbered from 0 through n, we delete the squares `'(0 . 0)` and cons (n, n) from the set of all squares of the original board. The function `delete` removes only the first occurrence of an element in a list.

```
DEFINITION:
delete ($x$, $l$)
=  if $l \simeq$ nil then $l$
   elseif $x$ = car ($l$) then cdr ($l$)
   else cons (car ($l$), delete ($x$, cdr ($l$))) endif
```

```
DEFINITION:
mutilated-board ($n$)
=  delete (cons ($n$, $n$), delete ('(0 . 0), make-all-rows ($n$, $n$)))
```

Examples of mutilated 2×2 and 8×8 boards are given below.

```
THEOREM: mutboard2
 mutilated-board (1) = '((0 . 1) (1 . 0))
```

```
THEOREM: mutboard7
 mutilated-board (7)
 =  '((0 . 1) (0 . 2) (0 . 3) (0 . 4) (0 . 5)
      (0 . 6) (0 . 7) (1 . 0) (1 . 1) (1 . 2)
      (1 . 3) (1 . 4) (1 . 5) (1 . 6) (1 . 7)
      (2 . 0) (2 . 1) (2 . 2) (2 . 3) (2 . 4)
      (2 . 5) (2 . 6) (2 . 7) (3 . 0) (3 . 1)
      (3 . 2) (3 . 3) (3 . 4) (3 . 5) (3 . 6)
      (3 . 7) (4 . 0) (4 . 1) (4 . 2) (4 . 3)
      (4 . 4) (4 . 5) (4 . 6) (4 . 7) (5 . 0)
      (5 . 1) (5 . 2) (5 . 3) (5 . 4) (5 . 5)
      (5 . 6) (5 . 7) (6 . 0) (6 . 1) (6 . 2)
      (6 . 3) (6 . 4) (6 . 5) (6 . 6) (6 . 7)
      (7 . 0) (7 . 1) (7 . 2) (7 . 3) (7 . 4)
      (7 . 5) (7 . 6))
```

Now, `all-covered-except-cornerp` can be defined to check a state for set equality with the set of squares of a mutilated board. The function `set-equal` checks for set equality by deleting common members of the given lists.

DEFINITION:
set-equal $(l1, l2)$
$= $ **if** $l1 \simeq$ **nil** **then** $l2 \simeq$ **nil**
  else $(\text{car}(l1) \in l2)$
    $\land$ set-equal (cdr $(l1)$, delete (car $(l1)$, $l2$)) **endif**

DEFINITION:
all-covered-except-cornerp $(x, n) =$ set-equal (x, mutilated-board (n))

We can now state that there does not exist a sequence of legal domino placement actions that when executed on an empty mutilated $n \times n$ $(n > 1)$ board will result in a state in which the board is completely covered.

THEOREM: tough-nut
(place-planp (p, n)
$\land$ $(n > 0)$
$\land$ ($s1$ = resultlist (p, **nil**))
$\land$ board-statep $(s1)$)
$\rightarrow$ ($\neg$ all-covered-except-cornerp $(s1, n)$)

The formal statement may be read as follows: If p is a plan for placing dominoes on a board numbered from 0 through n and $s1$ is the board state (not an error state) got by executing p on the empty board then all the squares of the board except the corner squares will not be covered in $s1$. Since all the variables are universally quantified from the outside, this says that there is no plan that can cover a mutilated board completely.

3.3 Proving the Theorem

If the above statement of the main theorem is provided as input to the theorem prover, can it prove it *automatically* using the impossibility argument involving the colors of squares? If not, how much assistance does it need from the user to find the proof? These are the questions we wish to discuss in this section. As it happens, the heuristics of the theorem prover are far, far too weak for it to find the proof by itself. The theorem prover requires considerable guidance from the user by way of definitions of new concepts and intermediate lemmas required for the proof. The main reasons for this are its inability to generate definitions of new concepts like colors, the lack of built-in arithmetic necessary for the proof and, most of all, its inability to choose a suitable generalization of a theorem that might be easier to prove by mathematical induction. However, once the necessary steps are supplied, the entire proof is checked reasonably efficiently by the theorem prover. It is worth noting that user guidance in the form of lemmas and definitions can never result in a proof of a non-theorem. That is, this form of user guidance is safe since the user cannot introduce facts that are not already provable.

We want the theorem prover to prove the main theorem as a consequence of the following two lemmas:

1. *In the state in which all squares of a mutilated $n \times n$ board are covered the number of covered white squares is less than the number of covered black squares.*

2. *If the number of covered white and black squares is equal in a state s of a $n \times n$ board and a sequence of legal domino placement actions is executed in s then the number of covered white and black squares would be equal in the resulting state.*

The theorem prover cannot get started on the proof because it does not "know" the concept of colors of squares. Colors are not provided as part of the problem statement and squares are simply pairs of numbers that give their position on the checkerboard. Thus, the lemmas cannot even be *stated* let alone proved. The theorem prover must somehow manufacture colors from the problem representation. Observe that the colors of the squares can be constructed from their coordinates. A square is white if its coordinates sum up to an even number and is black otherwise. The theorem prover has no heuristics for automatically generating new definitions. So we added the following definitions of predicates on white and black squares.

DEFINITION:
$\text{whitep}(x) = (((\text{car}(x) + \text{cdr}(x)) \ \mathbf{mod}\ 2) = 0)$

DEFINITION:
$\text{blackp}(x) = (((\text{car}(x) + \text{cdr}(x)) \ \mathbf{mod}\ 2) = 1)$

The lemmas are stated in terms of the number of black and white squares that are covered in a state. Therefore we need additional functions to compute the number of covered white and black squares in a state to express the lemmas. The functions `nwhite` and `nblack` compute the number of covered black and white squares in a state by going through the list of squares in a state and checking the sum of their coordinates.

DEFINITION:
```
nwhite ($x$)
=  if $x \simeq$ nil then 0
   elseif whitep (car ($x$)) then 1 + nwhite (cdr ($x$))
   else nwhite (cdr ($x$)) endif
```

DEFINITION:
nblack (x)

```
= if $x \simeq$ nil then 0
  elseif blackp (car ($x$)) then 1 + nblack (cdr ($x$))
  else nblack (cdr ($x$)) endif
```

Using these definitions we can now state the main lemmas as follows.

1. *In the state in which all squares of a mutilated $n \times n$ board are covered the number of covered white squares is less than the number of covered black squares.*

THEOREM: unequal3
$((n > 0) \land$ board-statep $(x) \land$ all-covered-except-cornerp $(x, n))$
$\rightarrow$ (nwhite $(x) <$ nblack (x))

2. *If the number of covered white and black squares is equal in a state s of a $n \times n$ board and a sequence of legal domino placement actions is executed in s then the number of covered white and black squares would be equal in the resulting state.*

THEOREM: bcequal1
(board-statep (s)
$\land$ (nwhite $(s) =$ nblack (s))
$\land$ place-planp (p, n)
$\land$ board-statep (resultlist (p, s)))
$\rightarrow$ (nwhite (resultlist (p, s)) $=$ nblack (resultlist (p, s))))

To understand the difficulty of carrying out a mechanical proof of the checkerboard solution, let us see what it takes to get the theorem prover to prove lemma 1. We may divide the proof into two parts, the proof of the lemma for boards with an odd number of rows and columns (given by lemma unequal1 below) and the proof of the lemma unequal3 for boards with an even number of rows and columns (given by lemma unequal2).

THEOREM: unequal1
(evenp (n)
$\land$ $(n > 0)$
$\land$ board-statep (x)
$\land$ all-covered-except-cornerp (x, n))
$\rightarrow$ (nwhite $(x) <$ nblack (x))

THEOREM: unequal2
(oddp $(n) \land$ board-statep $(x) \land$ all-covered-except-cornerp (x, n))
$\rightarrow$ (nwhite $(x) <$ nblack (x))

We shall discuss the difficulty of proving lemma unequal2 alone since it is a typical case. At first glance, it looks like the lemma may be proved by induction on n. But it is unclear (to us! and more so to the theorem prover) how the induction step can be carried out. However, it is clear that we can prove the lemma by first proving the following theorem that the number of black and white squares in an ordinary board with an even number of rows and columns (i.e. one in which the rows and columns are numbered from 0 to an odd number) are equal and then proving that deleting the corner squares makes the number of white squares less than the number of black squares because the corner squares are white.

THEOREM: eq-bw-board
oddp (n)
$\rightarrow$ (nwhite (make-all-rows $(n,\ n)$) = nblack (make-all-rows $(n,\ n)$))

Can the theorem prover find the proof of the above lemma? It is clear to us that the above theorem can be proved easily as a consequence of the following more general lemma that says that the number of black and white squares are equal in any board with an even number of columns irrespective of the number of rows in the board. The theorem prover cannot make such a generalization of a proposed conjecture; thus this generalized lemma given below must once again be proposed as an intermediate step by the user.

THEOREM: eq-bw-board1
oddp (n)
$\rightarrow$ (nwhite (make-all-rows $(m,\ n)$) = nblack (make-all-rows $(m,\ n)$))

The proof of this theorem is obvious because the number of white squares is equal to the number of black squares in every row of a board with an even number of columns. The total number of white and black squares in such a board are equal because they are equal in every row. Unfortunately, the theorem prover's heuristics are inadequate for it to determine this and so these facts must once again be supplied as lemmas to be proved. The lemma below says that the number of black and white squares in every row of a board is equal if there are an even number of columns.

THEOREM: equal-bw-row1
oddp (n) $\rightarrow$ (nwhite (make-row $(m,\ n)$) = nblack (make-row $(m,\ n)$))

This lemma can be proved by induction on n. The base case for $n = 1$ is shown below.

THEOREM: equal-bw-row-base0
nwhite (make-row $(m,\ 1)$) = nblack (make-row $(m,\ 1)$)

When $n = 1$, make-row $(m,\ \texttt{1})$ returns a list with the two squares of row m viz. cons $(m,\ \texttt{0})$ and cons $(m,\ \texttt{1})$. The number of white squares is equal to the number of black squares in this row because if the first square is white the second one would be black and vice versa. This follows from basic number theory since adding 1 to an odd number results in an even number and vice versa. Therefore, in any row of a checkerboard, a black square succeeds a white square and vice versa. The theorem prover cannot prove the above lemma by itself even when proposed by the user for various reasons. Because, the arguments of `make-row` include variables, the theorem prover's heuristics direct it to not expand the term but instead try induction on m. It cannot induct on m because `make-row` is defined recursively on its second argument. Thus, the proof fails forcing the user to direct it to expand the term by introducing the following lemma.

THEOREM: l1
make-row $(m,\ \texttt{1}) =$ list (cons $(m,\ \texttt{0})$, cons $(m,\ \texttt{1})$)

The theorem prover's built-in number theory axioms and its heuristics are not sufficient for proving the basic facts about odd and even numbers needed for the proof. Both the necessary facts about numbers and facts about black squares being adjacent to white squares in a row have to be formulated as lemmas and supplied by the user. The following lemmas say that odd numbers succeed even numbers and that even numbers succeed odd numbers.

THEOREM: t1
$((n \ \mathbf{mod}\ 2) = \texttt{0}) \rightarrow (((1 + n)\ \mathbf{mod}\ 2) = \texttt{1})$

THEOREM: t4
$((n \ \mathbf{mod}\ 2) = \texttt{1}) \rightarrow (((1 + n)\ \mathbf{mod}\ 2) = \texttt{0})$

The square next to a black square in a row is white.

THEOREM: sq-wb2
blackp (cons $(m,\ n)$) $\rightarrow$ whitep (cons $(m,\ 1 + n)$)

A square next to a white square in a row is black.

THEOREM: sq-wb1
whitep (cons $(m,\ n)$) $\rightarrow$ blackp (cons $(m,\ 1 + n)$)

Even more basic number theory facts are needed for proving some of the above lemmas such as the following.

THEOREM: t3
$(x \geq 2) \rightarrow ((1 + (x - 2)) = ((1 + x) - 2))$

The hypothesis $x \geq 2$ is needed because subtraction is defined on natural numbers and not integers.

THEOREM: plus1
$(m + (1 + n)) = (1 + (m + n))$

To prove the induction step of lemma eq-bw-row1, many facts about appending lists of squares are needed. For instance, we need to prove that the number of white/black squares in a list got by appending two lists of squares is equal to the sum of the number of white/black squares in the individual lists.

We shall stop our exposition at this point since we have a clear picture of the detailed level at which the theorem prover needs guidance from the user. For the rest of the proof, the reader is referred to the complete proof given in Appendix B.

3.4 Some Plans for Tiling Boards

Here are some simple plans to tile checkerboards and theorems that must be proved to verify them. The statements of the theorems and the examples on which they were tested are given in Section B.1. The proofs, however, haven't been carried out mechanically.

Consider a plan to cover a row numbered m completely with dominoes given that the row is empty in the initial state and that it has an even number of columns. This is a simple but ubiquitous problem. For instance, it can take the form of a program for filling or erasing a line on the screen. The simple plan to do this is a loop: place one tile after another on the row until it becomes fully covered. Using our method of expressing such program-like plans by plan generating programs, the plan to place tiles on a row from the first to the last column is given by `fill-row`.

```
DEFINITION:
fill-row ($m$, $n$)
=  **if** $n \simeq 0$ **then nil**
   **elseif** $n = 1$ **then** list (place (cons (cons ($m$, 0), cons ($m$, 1))))
   **else** append (fill-row ($m$, $n - 2$),
               list (place (cons (cons ($m$, $n - 1$), cons ($m$, $n$))))) **endif**
```

To state the correctness of `fill-row` for verifying the above plan, we need the predicates `empty` and `covered` that respectively check if the first n columns of row m are completely empty and completely covered.

DEFINITION:
empty $(m,\ n,\ s)$
$=$ **if** $n \simeq$ `0` **then** cons $(m,$ `0`$) \notin s$
  else (cons $(m,\ n) \notin s) \land$ empty $(m,\ n-1,\ s)$ **endif**

DEFINITION:
covered $(m,\ n,\ s)$
$=$ **if** $n \simeq$ `0` **then** cons $(m,$ `0`$) \in s$
  else (cons $(m,\ n) \in s) \land$ covered $(m,\ n-1,\ s)$ **endif**

The correctness theorem for `fill-row` along with an example are given below.

THEOREM: fill-row-ex0
fill-row (`2`, `3`)
$=$ `'((place ((2 . 0) 2 . 1))`
   `(place ((2 . 2) 2 . 3)))`

THEOREM: fill-row-ex1
$((s0 =$ `'((1 . 2) (1 . 3))`$)$
$\land$ $(p1 =$ fill-row (`2`, `3`)$)$
$\land$ $(s1 =$ resultlist $(p1,\ s0)))$
$\rightarrow$ (covered (`2`, `3`, $s1$) $\land$ place-planp ($p1$, `3`) $\land$ board-statep ($s1$))

†

THEOREM: fill-row-works
(board-statep ($s0$)
$\land$ oddp (n)
$\land$ $(m \in \mathbf{N})$
$\land$ empty $(m,\ n,\ s0)$
$\land$ $(p1 =$ fill-row $(m,\ n))$
$\land$ $(s1 =$ resultlist $(p1,\ s0)))$
$\rightarrow$ (covered $(m,\ n,\ s1)$ $\land$ place-planp $(p1,\ n)$ $\land$ board-statep ($s1$))

Another plan that arises in many contexts is the one to cover an $m \times n$ area completely. `Fill-board` is a plan generating program that covers a board whose rows are numbered from 0 to m and whose columns are numbered from 0 to n completely with dominoes. It uses `fill-row` to fill one row of the board after another until the board is completely covered. Its definition and a formal statement of its correctness are given below along with an example.

DEFINITION:
fill-board (m, n)
$=$ **if** $m \simeq$ `0` **then** fill-row (`0`, n)
else append (fill-board $(m - 1, n)$, fill-row (m, n)) **endif**

THEOREM: fill-board-ex0
fill-board (`2`, `3`)

```
= '((place ((0 . 0) 0 . 1))
    (place ((0 . 2) 0 . 3))
    (place ((1 . 0) 1 . 1))
    (place ((1 . 2) 1 . 3))
    (place ((2 . 0) 2 . 1))
    (place ((2 . 2) 2 . 3)))
```

THEOREM: fill-board-ex1
set-equal (make-all-rows (`2`, `3`), resultlist (fill-board (`2`, `3`), **nil**))

†

THEOREM: fill-board-works
(oddp $(n) \wedge (m \in \mathbf{N})$)
$\rightarrow$ set-equal (make-all-rows (m, n), resultlist (fill-board (m, n), **nil**))

Chapter 4

Formalizing Plan Constraints

So far we have only dealt with problems that prescribed two constraints to restrict the space of possible solutions, an initial condition and a goal condition. In fact, most work in commonsense reasoning [86, 61, 35] and imperative program synthesis [105, 65, 70, 72] deals only with such problems. However, as explained in Chapter 1, problems that specify many more constraints on solutions arise in many contexts particularly during program development. The additional constraints in a problem specification may be categorized as follows:

1. Time requirements. When constructing programs, on most occasions we require not just some solution to a problem but an efficient solution. For instance, to clear a block, one can use one of many possible solutions such as a plan that unstacks all towers or a plan that unstacks all blocks of the stack containing the block to be cleared. Typically, we want an efficient solution and sometimes the fastest or best solution. For the problem of clearing a block, we know that the plan specified by `makeclear-gen` is optimal in the number of actions needed. If all actions are assumed to take unit time, then the time taken to execute a plan is given by the number of actions in the plan. If distinct actions take different amounts of time, perhaps due to differences in their implementations, then the time taken by a plan depends on the actual actions chosen. In general, we may have costs associated with various types of actions and a problem may require finding a least cost solution.

2. Constraints on intermediate states and arbitrary sequences of intermediate states. Call the sequence of states generated by the execution of a plan p in state s the *history* or *execution sequence* generated by p in s. In addition to specifying a goal condition, a problem may specify properties that intermediate states brought about by the execution of a plan must satisfy. For instance, to minimize the number of new stacks created during the execution of a plan to clear a block b, we may try for a plan that satisfies the additional constraint that none of the stacks other than the one containing b be disturbed. That is, all the "other" stacks, the ones not containing b, must be present in every state in the execution sequence of any solution. A stronger constraint is to require that the number of stacks that exist in any state during the execution of a

solution must be no more than 1 + the number of stacks in the initial state. However, restrictions on execution sequences may also be combined with timing constraints. In general, a problem may specify various goals to be maintained during various sub-sequences of states or *intervals* in the history generated by a plan. For example, we may require a plan to clear a block b within 5 units of time and then keep b clear during the next 3 units.

3. Constraints on the actions in a plan. Sometimes restrictions on the kinds of actions allowed in a solution can be indirectly specified as constraints on states and vice versa. This is because actions result in a change in state and therefore constraints on what actions must be done can be indirectly specified as constraints on what state changes should or shouldn't take place and vice versa. For instance, the condition that none of the "other" stacks be disturbed while clearing a block can be enforced by restricting plans to contain the following kinds of actions: move actions that do not involve moving a block from or to the top of any of the "other" stacks and unstack actions that do not move the top of any of the "other" stacks with more than one block. Just as the above constraint is most naturally expressed in terms of execution sequences, so also are there conditions that are most naturally expressed as restrictions on the actions in a plan. Examples of such constraints are restrictions on the order in which actions are done and restrictions on the number of times a type of action occurs in a plan. For instance, we may stipulate that the plan to clear a block must contain a minimum number of unstack actions, or that every unstack action in it must be followed by one or more move actions. Restrictions on the number of times an action is done may also be specified: the block being cleared must be moved no more than n times or that a particular action must be executed for 5 units (i.e. 5 times in sequence).

4. Constraints requiring actions or action sequences to be executed whenever certain properties are satisfied by intermediate states. We could have constraints that combine the characteristics of the above 3 items. A problem may require that certain actions must or must not be executed in an intermediate state depending on the properties satisfied by the state. A simple example is the requirement that the plan to clear a block b must unstack b as soon as it becomes clear.

In this chapter, we will show that these constraints can be expressed as Lisp predicates on plans. To verify that a solution satisfies the given constraints, all we have to do is to prove that the output of plan generating programs satisfy the predicates corresponding to the constraints for all inputs. Our examples are problems from the blocks world such as the ones given above. We provide solutions to such problems and state theorems that must be proved to verify the solutions using our mechanized formalization. The theorems mentioned in this chapter have been tested on particular data but their mechanical proofs have not been carried out.

4.1 Related Work

While some planning systems deal with problems involving time and other forms of constraints, relatively little work in AI addresses general problems with arbitrary constraints. Vere [114] describes a planning system that accepts problems in propositional STRIPS [113] as input and generates a "partially ordered network of activities" as output. Fox [32] describes an expert system to plan and schedule tasks required for manufacturing products. More recently, Penberthy and Weld [103] describe a planner based on propositional STRIPS to generate plans that satisfy some temporal constraints. The main disadvantage of these systems is that general problems such as the problem of clearing a block cannot be specified as input. Some temporal logics have also been proposed in AI [3, 89, 87] but it is unclear how effective they are in practice.

Constraints on execution sequences generated by the actions of an agent also arise when specifying the behavior of *reactive* systems [44, 67]. A *reactive system* is one whose role is to maintain an ongoing, usually non-terminating, interaction with its environment. Examples of such systems are digital watches, vending machines, and microwave ovens. Most reactive systems involve concurrent actions. *Temporal logic* [67, 66, 26] based on *modal logic* has often been advocated as a formalism for specifying the behavior of reactive systems. In a temporal logic, explicit mention of states is avoided, i.e., there are no terms that stand for states. Instead temporal operators are used to describe properties of states that are reachable from a given state. At present, mechanization of temporal logic has mostly been confined to propositional temporal logic [26]. Propositional temporal logic is suitable for verifying properties of finite state systems [15] but not for specifying general problems. One other disadvantage of temporal logic is the inability to express time durations.

4.2 Our Approach

In this section, we describe how various constraints on solutions may be expressed as Lisp predicates on plans in the logic.

To express time requirements, we simply define a Lisp function from plans to plan durations (natural numbers) in the logic. Any restriction on plan duration may then be expressed as a Lisp predicate on plans that computes the duration of a given plan and checks if the duration satisfies the desired properties. If actions take unit time, then the length of a plan gives the amount of time taken by a plan. Otherwise Lisp functions that compute the time taken by a plan by doing a case analysis on the actions in the plan must be defined. Any proposed solution to a given problem must be proved to satisfy the Lisp predicates specifying the given temporal constraints in addition to proving that the plan brings about a goal state.

Constraints on actions belonging to a plan can also be specified as Lisp predicates on plans. Arbitrary predicates on plans can be defined by Lisp programs that go through a given plan and return **t** or **f** depending on whether or not the individual actions that occur in the plan are of the right kind.

Constraints on histories generated by plans can be specified as predicates on histories. Given a state s and a plan p, the execution sequence generated by p in s is represented as a list of states starting with s and ending with resultlist $(p,\ s)$. We define a program `make-hist` that takes a state s and a plan p as input and returns the history generated by p in s as output. Predicates on histories are specified by Lisp programs that check if the states belonging to a given history satisfy the desired properties. To express constraints on histories generated by plans, we simply define a predicate that computes the history generated by a given plan in a state and checks if it satisfies the desired properties. Constraints on actions belonging to plans that depend upon the history generated may also be checked by Lisp programs that take both plans and histories as arguments.

Since the additional constraints in a problem translate into additional predicates to be satisfied by a proposed solution, no changes to our representation of plans in the logic need be made. We merely have to prove that plans generated by programs defined in the logic satisfy the predicates corresponding to the various constraints specified in a problem.

In the rest of the chapter, we give examples of problems with constraints and plans that solve them and state theorems that must be proved to verify the plans. The statements of the theorems and the examples on which they were tested are given in Section A.3. The mechanical proofs of the theorems were not carried out.

4.3 Time

Suppose our problem is to find an optimal solution (in the number of actions needed) to the problem of clearing a block. We know that our program `makeclear-gen` specifies an optimal solution since any plan to solve the problem must move all the blocks on top of the block being cleared at least once. This is stated using our formalization of the blocks world given in Appendix A as follows. †

THEOREM: c4-th1
(bw-statep (s)
$\land$ find-stack-of-block $(b,\ s)$
$\land$ planp (p)
$\land$ $(s1 =$ resultlist $(p,\ s))$
$\land$ bw-statep $(s1)$
$\land$ clear $(b,\ s1))$
$\rightarrow$ (len (makeclear-gen $(b,\ s)) \leq$ len $(p))$

To prove the theorem we must show that the number of blocks above the block being cleared is decreased by at most one by any action in the blocks world and that `makeclear-gen` generates exactly as many actions as the number of blocks above b.

Here is an example of how we might prove optimality even when move and unstack actions take different amounts of time. Suppose we want to prove that the plan specified by `makeclear-gen` is optimal when move actions take more time than unstack actions. To express the theorem, we first define a function `plan-duration` that computes the time taken by a plan using the times taken by the move and unstack actions.

```
DEFINITION:
plan-duration ($p$, $move\text{-}time$, $unstack\text{-}time$)
=  **if** $p \simeq$ **nil then** 0
   **elseif** movep (car ($p$))
   **then** $move\text{-}time$ + plan-duration (cdr ($p$), $move\text{-}time$, $unstack\text{-}time$)
   **else** $unstack\text{-}time$ + plan-duration (cdr ($p$),
                                               $move\text{-}time$,
                                               $unstack\text{-}time$) **endif**
```

The following theorem shows how `plan-duration` works.

THEOREM: plan-dur1
plan-duration (`'((unstack a) (move b c) (move a b))`, 2, 1)
$=$ 5

Below we state that the plan specified by `makeclear-gen` is optimal in the time taken if the time to unstack blocks (*unstack-time*) is less than the time to move blocks (*move-time*). †

THEOREM: c4-th2
(bw-statep (s)
$\wedge$ find-stack-of-block (b, s)
$\wedge$ planp (p)
$\wedge$ ($s1$ = resultlist (p, s))
$\wedge$ bw-statep ($s1$)
$\wedge$ clear (b, $s1$)
$\wedge$ ($unstack\text{-}time < move\text{-}time$))
$\rightarrow$ (plan-duration (makeclear-gen (b, s), $move\text{-}time$, $unstack\text{-}time$)
$\leq$ plan-duration (p, $move\text{-}time$, $unstack\text{-}time$))

The important point here is that the function to compute plan durations may be specified as a Lisp program that may take any number of parameters. Thus, more complicated functions for computing plan durations can be easily formalized. Sometimes it is even possible to prove the *complexity* of an algorithm for a particular class of problems. For instance, we can prove that the number of actions in makeclear-gen $(b,\ s)$ is proportional to the number of blocks in the stack containing b in s. That is, `makeclear-gen` is $O(n)$, where n is the number of blocks of the stack containing b in s. †

THEOREM: c4-com1
(bw-statep (s) $\wedge$ find-stack-of-block $(b,\ s)$)
$\rightarrow$ (len (makeclear-gen $(b,\ s)$) $\leq$ len (find-stack-of-block $(b,\ s)$))

4.4 Constraints on Execution Sequences

Suppose we want a plan to clear a block b without disturbing any of the stacks that do not contain b. We can state this as a constraint on execution sequences by insisting that all the stacks other than the one containing b belong to every state in the execution sequence of a plan. Below we show how this can be done. The function `make-hist` constructs the history got by executing a plan p in state s.

DEFINITION:
make-hist $(s,\ p)$
= **if** $p \simeq$ **nil then** cons $(s,$ **nil**$)$
  else cons $(s,$ make-hist (result (car (p), s), cdr (p))) **endif**

Here is an example of what `make-hist` produces. Consider the blocks world state STATE1 in which there are two stacks, one with blocks `'a`, `'b` and `'c` and the other with blocks `'d` and `'e`.

DEFINITION: STATE1 = `'((a b c) (d e))`

The plan to clear block `'c` in STATE1 generated by `makeclear-gen` is `'((unstack a) (unstack b))`.

THEOREM: makeclear-ex2
makeclear-gen (`'c`, STATE1) = `'((unstack a) (unstack b))`

The history generated by this plan when executed in STATE1 is given below.

THEOREM: hist-ex1
make-hist (STATE1, makeclear-gen (`'c`, STATE1))
= `'(((a b c) (d e))`
   `((b c) (a) (d e))`
   `((c) (b) (a) (d e)))`

The function `other-stacks` returns the list of stacks in a state s that do not contain the block b.

DEFINITION:
other-stacks (b, s) = delete (find-stack-of-block (b, s), s)

THEOREM: other-stacks-ex1
other-stacks (`'c`, STATE1) = `'((d e))`

To ensure that none of the other stacks are disturbed during the execution of a plan, we define `check-other-stacks` that takes a history h and a list of stacks l as arguments and checks if every state in h contains every stack in l.

DEFINITION:
subset ($s1$, $s2$)
= **if** $s1 \simeq$ **nil then t**
   else (car ($s1$) $\in$ $s2$) $\wedge$ subset (cdr ($s1$), $s2$) **endif**

DEFINITION:
check-other-stacks (l, h)
= **if** $h \simeq$ **nil then t**
   else subset (l, car (h)) $\wedge$ check-other-stacks (l, cdr (h)) **endif**

We can use `check-other-stacks` to express the constraint that none of the states other than the one containing `'c` in STATE1 is disturbed in the history generated by executing makeclear-gen (`'c`, STATE1) in STATE1 as follows.

THEOREM: check-other-ex1
check-other-stacks (other-stacks (`'c`, STATE1),
          make-hist (STATE1, makeclear-gen (`'c`, STATE1)))

More generally, we can state that if a block b exists in state s, then the plan makeclear-gen (b, s) does not disturb the stacks in s that do not contain b during execution. †

THEOREM: c4-th3
(bw-statep (s)
$\wedge$ find-stack-of-block (b, s)
$\wedge$ (p = makeclear-gen (b, s)))
$\rightarrow$ check-other-stacks (other-stacks (b, s), make-hist (s, p))

A problem may also stipulate goals that require that certain conditions be maintained for some interval of time. Suppose we want a plan that would bring about some interval (sequence of states) I of duration $\geq n$ in the future of the initial state such that b is clear in every state in I. We can achieve this goal by clearing b and then executing the action unstack (b) n times in succession. The function `unstackn` generates a sequence of n unstack (b) actions.

DEFINITION:
```
unstackn (b, n)
= if n ≃ 0 then nil
  else cons (unstack (b), unstackn (b, n − 1)) endif
```

`Make-clear-intn` is the composite plan for achieving the goal of keeping b clear for at least n units of time after b becomes clear.

DEFINITION:
make-clear-intn (b, n, s) = append (makeclear-gen (b, s), unstackn (b, n))

For example, `make-clear-intn` generates the following plan to keep `'c` clear for 2 units of time in the future of STATE1.

THEOREM: clear-intn-ex1
```
make-clear-intn ('c, 2, STATE1)
= '((unstack a)
    (unstack b)
    (unstack c)
    (unstack c))
```

The predicate `check-clearn` checks if block b is clear for at least n units of time, i.e., in the the first *n+1* states, of the history h.

DEFINITION:
```
check-clearn (b, h, n)
= if h ≃ nil then f
  elseif n ≃ 0 then clear (b, car (h))
  else clear (b, car (h)) ∧ check-clearn (b, cdr (h), n − 1) endif
```

The function `exist-clear-intn` checks if there is a subsequence of length n in the history h in which the block b is clear.

DEFINITION:
exist-clear-intn (b, h, n)
= **if** $h \simeq$ **nil then f**
  else check-clearn (b, h, n) $\vee$ exist-clear-intn (b, cdr (h), n) **endif**

The following theorems test the behavior of `exist-clear-intn`.

THEOREM: hist-ex2
make-hist (STATE1, make-clear-intn (`'c`, `2`, STATE1))

```
= '(((a b c) (d e))
    ((b c) (a) (d e))
    ((c) (b) (a) (d e))
    ((c) (b) (a) (d e))
    ((c) (b) (a) (d e)))
```

DEFINITION:
H1 = make-hist (STATE1, make-clear-intn (`'c`, `2`, STATE1))

THEOREM: ex-ex1
exist-clear-intn (`'c`, H1, `2`)

We can state the general theorem that make-clear-intn (b, n, s) achieves the goal of keeping b clear for an interval of n units of time in the future of the initial state s as follows. †

THEOREM: clear-int-th1
(bw-statep (s)
$\wedge$ find-stack-of-block (b, s)
$\wedge$ ($n \in \mathbf{N}$)
$\wedge$ (p = make-clear-intn (b, n, s))
$\wedge$ (h = make-hist (s, p)))
$\rightarrow$ exist-clear-intn (b, h, n)

4.5 Constraints on Actions

Suppose we want to clear a block b without moving it. Then, we must check if the plan generated as a solution to the problem does not move b. This can be specified as a predicate to be satisfied by plans generated to solve the problem. Observe that the first argument of the move and unstack actions is the object that is moved. Therefore, all we have to do is to check that the first argument of all the actions in a plan is not equal to b. More generally, we can define a function that computes the number of times a particular block is moved when executing a plan. Using this function, any constraints on the number of times a block is moved can be expressed.

```
DEFINITION:
number-of-times-moved ($p$, $b$)
=  if $p \simeq$ nil then 0
   elseif cadar ($p$) = $b$ then 1 + number-of-times-moved (cdr ($p$), $b$)
   else number-of-times-moved (cdr ($p$), $b$) endif
```

THEOREM: number-ex1
number-of-times-moved ('((unstack a) (unstack b)), 'c) = 0

THEOREM: number-ex2
number-of-times-moved ('((unstack a) (unstack b)), 'a) = 1

We can state that the plan specified by `makeclear-gen` does not move the block being cleared as follows. †

THEOREM: ch4-th5
(bw-statep (s) $\wedge$ find-stack-of-block (b, s))
$\rightarrow$ (number-of-times-moved (makeclear-gen (b, s), b) = 0)

Here is an example of a constraint on the order in which actions occur in a plan. We can check if every unstack action in a plan p is followed by a move action using the following predicate.

```
DEFINITION:
move-follows-unstackp ($p$)
=  if ($p \simeq$ nil) $\vee$ (cdr ($p$) $\simeq$ nil) then t
   elseif unstackp (car ($p$))
   then movep (cadr ($p$)) $\wedge$ move-follows-unstackp (cddr ($p$))
   else move-follows-unstackp (cdr ($p$)) endif
```

Clearly this constraint is not satisfied by the plan specified by `makeclear-gen`.

THEOREM: move-follows-ex1
$\neg$ move-follows-unstackp (makeclear-gen (`'c`, STATE1))

THEOREM: move-follows-ex2
move-follows-unstackp (`'((move a b) (unstack c) (move b c))`) †

THEOREM: ch4-th6
(bw-statep (s) $\wedge$ find-stack-of-block (b, s))
$\rightarrow$ ($\neg$ move-follows-unstackp (makeclear-gen (b, s)))

Our final example involves checking both the execution sequence generated by a plan as well as the actions composing a plan. This is needed if we require a plan to clear a block b that unstacks b as soon as it becomes clear. The predicate `check-both` takes a history h, a block b and a plan p as arguments and checks if an unstack action is executed in the state in which b becomes clear.

DEFINITION:
check-both (h, p, b)

```
= if $h \simeq$ nil then f
  elseif clear ($b$, car ($h$)) then car ($p$) = unstack ($b$)
  else check-both (cdr ($h$), cdr ($p$), $b$) endif
```

Here is a simple example to test `check-both`.

DEFINITION:
P2 = append (makeclear-gen (`'c`, STATE1), list (unstack (`'c`)))

DEFINITION: H2 = make-hist (STATE1, P2)

THEOREM: check-both-ex1
check-both (H2, P2, `'c`) †

THEOREM: ch4-th7
(bw-statep (s)
$\wedge$ find-stack-of-block (b, s)
$\wedge$ (p = append (makeclear-gen (b, s), list (unstack (b))))
$\wedge$ (h = make-hist (s, p)))
$\rightarrow$ check-both (h, p, b)

Chapter 5

A General Framework

In the preceding chapters, we illustrated how general problems and problem domains such as the blocks world can be specified by writing interpreters in the Boyer-Moore logic. We also showed how plan generating programs, programs that produce a sequence of actions depending on the initial state and other parameters given as input, can be used to express and verify mechanically solutions to general problems. While our approach does overcome some of the drawbacks of the existing methods such as the need for a large number of explicit frame axioms and the inability to specify actions with side-effects, it does suffer from the following disadvantages:

1. Right now, each problem domain is formalized by inventing a new "programming language" depending on the data structures chosen to represent the states and actions of the system. By choosing a uniform representation for actions and states over all problem domains, i.e., by developing a single programming language or machine for all problem domains and formalizing its semantics, we gain a number of important advantages. First, theorems proved about states and actions can be re-used across domains. Thus, it would be possible to build a library with basic oft-used actions (programs) and state data structures (data types) formalized and made available for problem solving. Since solutions to distinct problems are programs (plans) of the same machine, we might be able to use solutions of problems solved in the past as subprograms (sub-plans) when solving a new problem. This is particularly important from the standpoint of program development. Secondly, the addition of new actions to an existing formalization can be done without any change to the interpreter. For instance, when we wanted to add a new action `move-over` to our blocks world formalization in Chapter 2, we changed our programming language and were therefore forced to re-define the interpreter for the new machine. This can be avoided if we use a single programming language for specifying all domains.

2. So far, plans only consist of primitive actions. It would desirable if we could include plans such as the plan for clearing a block as atomic *complex actions* in larger plans much the way procedures are combined in an imperative programming language. For this, we need terms in the logic for representing such

complex actions. The ability to "compose" plans like procedures will let us describe large, complex plans succinctly. This would be particularly useful when solutions to problems are used across problem domains as contemplated above.

3. So far, we have assumed that all the effects of all actions in a domain are given. Sometimes, it may be necessary to specify *partial actions*[1] wherein the effects of actions on only some aspects of the state may be given. In such a case, it would not be possible to define an interpreter for the system as we have been doing since actions cannot be specified by programs.

In this chapter, we present solutions to overcome these disadvantages. Our first step is to represent primitive actions by data structures that stand for the *names* of the functions that specify them. Thus, if the action of moving block x to the top of block y is specified by the function res-move$(x,\ y,\ s)$, then the action is represented by a data structure that stands for $\lambda s.$ res-move$(x,\ y,\ s)$ as in [76]. Such terms are called *fluents* by McCarthy [76, 86]. The data structures that represent such lambda expressions are chosen so that `result` can evaluate lambda expressions on particular states using the Lisp interpreter `eval$` available as a function in the logic. For instance, the action of moving `'a` to the top of `'b` specified by the function `res-move` in Chapter 2 is now represented by the data structure `'(res-move 'a 'b)` that stands for $\lambda s.$ res-move(`'a`, `'b`, s). To execute the action `'(res-move 'a 'b)` in a state *s0*, `result` constructs the term corresponding to the description or *quotation* of the term res-move(`'a`, `'b`, *s0*) and evaluates it using `eval$` to obtain the value of res-move(`'a`, `'b`, *s0*). Such quotations of function calls to action specifications are already available in the logic because the meta-theory of the Boyer-Moore logic is formalized as part of the logic, i.e., for every term in the logic there exists a term that is its syntactic description or quotation. Notice that we do not represent the bound "lambda variable" explicitly in the representation of actions. Instead we adopt the convention that the last argument of a function that specifies an action is the initial state argument.

This device of representing actions as fluents can be used for representing both primitive and complex actions. For instance, suppose the input-output behavior of the plan given by the plan generator `makeclear-gen` in Chapter 2 is specified by a function `res-makeclear` which takes a block b and a state s as parameters and returns the state got by executing the plan generated by `makeclear-gen` in s. Then the complex action corresponding to the plan given by `makeclear-gen` can once again be represented as $\lambda s.$ res-makeclear$(b,\ s)$. Since *result* and *resultlist* can once again be used to execute such actions in various states by lambda evaluation, we may allow

[1]Sometimes the phrase "partial action" is used to talk about actions that cannot be executed at all times. We will, however, use the phrase to describe an action whose effects on a state can only be given by a partially specified function.

complex actions to be included in plans. Thus, with our new representation, a plan in a domain is a sequence of actions where each action may be either a primitive action or a complex action.

Because all actions of all domains are represented in the same way, we now have a single machine or "programming language" whose states and actions encompass those of every other problem domain. The set of states of this general machine is given by the set of Lisp data structures and the set of actions is given by those data structures that can be interpreted as lambda expressions that are names of action specifications. Plans, sequences of actions, may be thought of as programs of this machine and the interpreter *resultlist* may be thought of specifying the operational semantics. We represent error states of this machine by lists whose car is `'failed`.

We propose to tackle the problem of specifying partial actions in the Boyer-Moore logic by specifying them using axioms introduced via the `CONSTRAIN` event [8]. The `CONSTRAIN` event allows us to add an axiom involving new function symbols using as "witnesses" or "models" previously defined functions. Thus, it is possible to use a subset of theorems about a completely defined Lisp program (action specification) to constrain a new function symbol that specifies an action partially. `CONSTRAIN` also guarantees consistency of axioms added. Naturally, actions specified partially cannot be executed on concrete data structures. It so happens that actions specified via `CONSTRAIN` cannot be evaluated by the interpreter `eval$` because function symbols introduced via `CONSTRAIN` do not have any "code" associated with them. Thus, such actions cannot be viewed as part of the programming language of our general machine. In the last section, we propose a simple way to augment the `CONSTRAIN` facility that might allow even partially specified actions to be evaluated by `eval$`.

5.1 The General Framework

We will first describe briefly how the syntactic forms of terms in the logic are encoded by other terms, their quotations, and explain how `eval$` works. Then we will present our general framework.

5.1.1 Definition of eval$

For a fuller description of the concepts presented in this section, the reader must consult [10]. The quotations of terms in the logic are formed as follows. Variables like x and function symbols like *fn* are encoded by the corresponding litatoms such as `'x` and `'fn`. Quotations of terms that are function applications are given by a list whose car is the quotation of the function symbol and whose cdr is a list of quotations of the individual arguments. For example, the quotation of $x + y$ is the term list (`'plus`, `'x`, `'y`) which is abbreviated as `'(plus x y)`. The quotation of $x +$

$(y + z)$ is list (`'plus`, `'x`, `'(plus y z)`) abbreviated as `'(plus x (plus y z))`. Quotations of a special class of terms $T1$ called *explicit value terms* are also represented by list(`'quote`,$T1$). Roughly speaking, explicit value terms correspond to the Lisp data structures of the logic. Thus, another quotation of cons (`'a`, cons (`'b`, **nil**)) is list (`'quote`, cons (`'a`, cons (`'b`, **nil**))).

In the definition of `eval$` given below, the function `apply$` is used. Rather than giving the exact definition of `apply$`, we will merely explain intuitively what it computes.

```
DEFINITION:
eval$ (flg, x, a)
=  if flg = 'list
   then if x ≃ nil then nil
        else cons (eval$ (t, car (x), a), eval$ ('list, cdr (x), a)) endif
   elseif litatom (x) then cdr (assoc (x, a))
   elseif x ≃ nil then x
   elseif car (x) = 'quote then cadr (x)
   else apply$ (car (x), eval$ ('list, cdr (x), a)) endif
```

`Eval$` takes three arguments, *flg*, x and a. If *flg* equals `'list`, `eval$` interprets x as a list of quotations, otherwise it processes x as a quotation of a particular term. The argument a is an "environment" or association list that assigns values to quoted variables, i.e., litatoms. Essentially, `eval$` "unquotes" a quoted term (or list of terms) x and returns its value (or list of values) using the values of the variables specified by a. For instance, given `t`, `'(plus x y)` and an environment `'((x . 2) (y . 3))` as arguments, `eval$` would evaluate x to 2 and y to 3 and use `apply$` to apply the definition of `plus` to list (2, 3) to get 5. `Apply$` takes a litatom that is a quotation of a function symbol *fn* and a list of values *args* as arguments and returns the value got by applying the definition of *fn* to the arguments given by *args*. The following simple examples show how `eval$` and `apply$` work on concrete data.

THEOREM: ch5-ex1
eval$ (`'list`, list (`'x`, `'y`), `'((x . 2) (y . 3))`) = `'(2 3)`

THEOREM: ch5-ex2
apply$ (`'plus`, `'(2 3)`) = `5`

THEOREM: ch5-ex3
eval$ (`t`, `'(plus x y)`, `'((x . 2) (y . 3))`) = `5`

THEOREM: ch5-ex4
eval$ (`t`, `'(plus (plus x y) (plus x y))`, `'((x . 2) (y . 3))`)
= `10`

It is important to keep in mind that quotations are also data structures (terms). Therefore, we can define a function that constructs and returns distinct quotations depending on the arguments passed to it. To understand how we represent parameterized actions (i.e. parameterized lambda expressions) using "parameterized" quotations and evaluate them using `eval$`, it is helpful to look at the following simple example.

DEFINITION:
append-template (x, y)
= list (`'append`, list (`'quote`, x), list (`'quote`, y))

The lambda variable in actions is not represented in the quotation since it is assumed to be the last argument by convention. Thus, parameterized actions are represented by quotations such as the one given above. `Append-template` constructs distinct quotations depending on the arguments passed to it. The second and third elements of the list returned are list (`'quote`, x) and list (`'quote`, y) rather than x and y because we do not want x and y to be evaluated by `eval$` when a term of the form append-template (x, y) is passed to it. In other words, we would like eval$ (**t**, append-template (`'(a b)`, `'(c d)`), **nil**) to be equal to append (`'(a b)`, `'(c d)`) whose value is `'(a b c d)`. Notice, however, that eval$ (**t**, list (`'append`, `'(a b)`, `'(c d)`), **nil**) would require evaluation of eval$ (**t**, `'(a b)`, **nil**) whose value is unpredictable because it depends on whether a function named `a` has already been defined or not.

5.1.2 Our Formal Theory

As explained earlier, actions are represented by data structures that stand for the names of functions that specify them. Thus, the representation of an action specified by the Lisp function res-fn(t1,t2,...,tn,s) is λs. res-fn(t1,t2,...,tn,s). The data structure representing this lambda expression is the quotation of the term res-fn(t1,t2,...,tn). The lambda variable is suppressed in the action data structures because it is assumed by convention to be the last argument of the function that specifies the action. Consequently, actions are specified by Lisp programs that take a state term as their last argument[2]. This gives us a uniform representation of both primitive and complex actions of all domains.

[2]If the specification of an action is independent of the state in which it is executed then we do not need a redundant state argument. This is because `apply$` ignores extra arguments when computing the value of a function.

As an example, recall the function `res-move` that we defined in Chapter 2 to specify the action of moving a block *b1* to the top of another block *b2* in a state *s* in the blocks world.

DEFINITION:
res-move (*b1*, *b2*, *s*)
= **if** (*b1* = *b2*) ∨ (¬ clear (*b1*, *s*)) ∨ (¬ clear (*b2*, *s*))
  then list (`'failed`, `'res-move`, *s*)
  else exec-move (*b1*, *b2*, *s*) **endif**

In our general framework, the action move (*b1*, *b2*) would be represented by the term list (`'res-move`, list (`'quote`, *b1*), list (`'quote`, *b2*)). In general, if res-fn(x1,x2,...,xn,s) specifies an action λs. res-fn(x1,x2,...,xn,s) then the action is represented by the term list (`'res-fn`, list (`'quote`, *x1*), list (`'quote`, *x2*),..., list (`'quote`, *xn*)).

The function *result* in our general theory computes the new state got by executing an action of the form list (`'res-fn`, list (`'quote`, *x1*), list (`'quote`, *x2*),..., list (`'quote`, *xn*)) in a state *s* by first constructing a term *term* equal to list (`'res-fn`, list (`'quote`, *x1*), list (`'quote`, *x2*), ...,list (`'quote`, *xn*), list (`'quote`, *s*)) by appending list (list (`'quote`, *s*)) to the given action term. Then, *result* returns eval$ (`t`, *term*, **nil**) which evaluates to res-fn(x1,x2,...,xn,s), the state, possibly an error state, got as a result of executing the given action in *s*. Thus, *result* essentially evaluates the lambda expression given by an action on a state. The definition of *result* in our general theory follows.

DEFINITION:
result (*a*, *s*)
= **if** car (*s*) = `'failed` **then** *s*
  else eval$ (`t`, append (*a*, list (list (`'quote`, *s*))), **nil**) **endif**

The following simple example illustrates how *result* operates.

DEFINITION: STATE1 = `'((a b c) (d e))`

THEOREM: gen-ex1
result (list (`'res-move`, list (`'quote`, 'b1`), list (`'quote`, 'b2`)),
    STATE1)
= res-move (`'b1`, `'b2`, STATE1)

Observe that the definition of `result` takes into account our convention that the last argument of all functions specifying actions be a state argument. Error states of our machine are lists whose car is the litatom `'failed`. It follows from the definition of `result` that the execution of any action in an error state results in the error state. That is, once the machine is in an error state it stays that way.

To prevent the user from having to wade through quotations representing action terms, we provide functions to construct, analyze and recognize actions. The function `make-action` constructs an action term when given as input, a litatom such as `'res-move` that is the quotation of a function symbol and a list of the action's arguments. The auxiliary function `make-arglist` returns a list of quoted terms corresponding to the arguments of an action.

```
DEFINITION:
make-arglist (al)
=  if al ≃ nil then nil
   else cons (list ('quote, car (al)), make-arglist (cdr (al))) endif
```

```
DEFINITION:
make-action (type, arglist) = cons (type, make-arglist (arglist))
```

For instance, the constructor for the move and the unstack actions of the blocks world can be easily defined using `make-action` without worrying about quotations.

```
DEFINITION:
move (b1, b2) = make-action ('res-move, list (b1, b2))
```

```
DEFINITION:
unstack (b) = make-action ('res-unstack, list (b))
```

In short, `make-action` constructs parameterized lambda expressions corresponding to actions.

Similarly, we provide a predicate `check-action` that can be used to recognize actions. `Check-action` takes an action term a and a litatom that is the quotation of a function symbol at and checks if a is a lambda expression formed using the function at. We do not check if the arguments of a are well-formed because they are taken care of by the preconditions of the action.

```
DEFINITION:  check-action (a, at) = (car (a) = at)
```

The predicates on move and unstack actions can be defined as follows.

```
DEFINITION:  movep (x) = check-action (x, 'res-move)
```

DEFINITION: unstackp(x) = check-action$(x$, `'res-unstack`)

To obtain the nth argument of an action we may use the following function `argn`. `Argn` takes into account that all arguments x of an action are represented as list(`'quote`, x). Thus, `argn` may be used to define predicates on actions that depend on the values of arguments without worrying about the underlying representation.

The function `nth` returns the nth value in a list whose elements are numbered starting with 0.

DEFINITION:
```
nth (n, l)
=  if n ≃ 0 then car (l)
   else nth (n − 1, cdr (l)) endif
```

DEFINITION: argn(n, a) = cadr(nth(n, a))

THEOREM: gen-ex2
argn(`1`, move(`'a`, `'b`)) = `'a`

Plans are once again represented as sequences of actions and the interpreter *resultlist* for the general machine is defined as before.

DEFINITION:
```
resultlist (l, s)
=  if l ≃ nil then s
   else resultlist (cdr (l), result (car (l), s)) endif
```

Plans may include both primitive and complex actions. In the next section, we explain how we represent complex actions as terms in the logic so that they may be used just like atomic actions in other plans.

We can use lambda expressions to also represent other kinds of fluents such as *propositional fluents* that are predicates on states as in [76, 72] and define the celebrated *holds* predicate [59, 61] in the same way we defined *result*. For instance, corresponding to predicates `find-stack-of-block` and `on` from Chapter 2, we can form fluent terms λs. find-stack-of-block(b, s) and λs. on (b1 b2 s). To understand the representation of these fluents as data structures, recall the definitions of `find-stack-of-block` and `on`.

DEFINITION:
```
find-stack-of-block (b, s)
=  if s ≃ nil then f
   elseif b ∈ car (s) then car (s)
   else find-stack-of-block (b, cdr (s)) endif
```

DEFINITION:
on $(b1,\ b2,\ s)$ = consecp $(b1,\ b2,$ find-stack-of-block $(b1,\ s))$

The definitions of `make-fluent` and `holds` defined below mirror those of `make-action` and `result`.

DEFINITION:
make-fluent $(type,\ arglist)$ = cons $(type,$ make-arglist $(arglist))$

DEFINITION:
holds $(x,\ s)$ = eval\$ (`t`, append $(x,$ list (list (`'quote`, s))), **nil**)

Here are a few examples to illustrate how the *holds* predicate works.

DEFINITION:
on-fluent $(b1,\ b2)$ = make-fluent (`'on`, list $(b1,\ b2)$)

DEFINITION:
find-stack-fluent (b)
= make-fluent (`'find-stack-of-block`, list (b))

THEOREM: holds-ex1
holds (on-fluent (`'b`, `'c`), STATE1)

In the Appendix C, we have given the definitions of the general theory followed by a formalization of the blocks world done using the general framework. The new blocks world formalization differs from the formalization in Appendix A only in the definitions of `move`, `unstack`, `movep` and `unstackp`. Of course, the definitions of `result` and `resultlist` are borrowed from the general theory.

A word about mechanical theorem proving in this general framework. Mechanical theorem proving is known to be very difficult in the presence of quotations and `eval$`. However, one way to substantially reduce the theorem proving effort is to prove theorems that can be used as rewrite rules to eliminate the occurrence of terms with `eval$` once and for all. In our theory, `eval$` occurs only in the definition of `result`. (We have not proved general theorems using `holds`.) Thus, we could prove the following theorems about `result` flagging them as rewrite rules and then disable the definition of `result` so that terms with `eval$` do not occur when proving other theorems.

THEOREM: result1
(bw-statep (s)
$\land$ $((\neg$ clear $(b1,\ s)) \lor (b1 = b2) \lor (\neg$ clear $(b2,\ s))))$
$\rightarrow$ (result (move $(b1,\ b2),\ s)$ = list (`'failed`, `'res-move`, s))

THEOREM: result5

$$\begin{aligned}
&(\text{bw-statep}\,(s)\\
&\ \wedge\ \ \text{clear}\,(\mathit{b1},\ s)\\
&\ \wedge\ \ \text{clear}\,(\mathit{b2},\ s)\\
&\ \wedge\ \ (\mathit{b1} \neq \mathit{b2})\\
&\ \wedge\ \ (\text{cdr}\,(\text{assoc}\,(\mathit{b1},\ s)) \simeq \mathbf{nil}))\\
&\to\ \ (\text{result}\,(\text{move}\,(\mathit{b1},\ \mathit{b2}),\ s)\\
&\qquad = \ \text{cons}\,(\text{cons}\,(\mathit{b1},\ \text{assoc}\,(\mathit{b2},\ s)),\\
&\qquad\qquad\quad \text{delete}\,(\text{assoc}\,(\mathit{b1},\ s),\ \text{delete}\,(\text{assoc}\,(\mathit{b2},\ s),\ s))))
\end{aligned}$$

THEOREM: result6

$$\begin{aligned}
&(\text{bw-statep}\,(s)\\
&\ \wedge\ \ \text{clear}\,(\mathit{b1},\ s)\\
&\ \wedge\ \ \text{clear}\,(\mathit{b2},\ s)\\
&\ \wedge\ \ (\mathit{b1} \neq \mathit{b2})\\
&\ \wedge\ \ \text{listp}\,(\text{cdr}\,(\text{assoc}\,(\mathit{b1},\ s))))\\
&\to\ \ (\text{result}\,(\text{move}\,(\mathit{b1},\ \mathit{b2}),\ s)\\
&\qquad = \ \text{cons}\,(\text{cons}\,(\mathit{b1},\ \text{assoc}\,(\mathit{b2},\ s)),\\
&\qquad\qquad\quad \text{cons}\,(\text{cdr}\,(\text{assoc}\,(\mathit{b1},\ s)),\\
&\qquad\qquad\qquad\quad \text{delete}\,(\text{assoc}\,(\mathit{b1},\ s),\ \text{delete}\,(\text{assoc}\,(\mathit{b2},\ s),\ s)))))
\end{aligned}$$

THEOREM: result2

$$\begin{aligned}
&(\text{bw-statep}\,(s) \wedge (\neg\ \text{clear}\,(b,\ s)))\\
&\to\ \ (\text{result}\,(\text{unstack}\,(b),\ s) = \text{list}\,(\texttt{'failed},\ \texttt{'unstack},\ s))
\end{aligned}$$

THEOREM: result3

$$\begin{aligned}
&(\text{bw-statep}\,(s) \wedge \text{clear}\,(b,\ s) \wedge (\text{cdr}\,(\text{assoc}\,(b,\ s)) \simeq \mathbf{nil}))\\
&\to\ \ (\text{result}\,(\text{unstack}\,(b),\ s) = \text{cons}\,(\text{list}\,(b),\ \text{delete}\,(\text{assoc}\,(b,\ s),\ s)))
\end{aligned}$$

THEOREM: result4

$$\begin{aligned}
&(\text{bw-statep}\,(s) \wedge \text{clear}\,(b,\ s) \wedge \text{listp}\,(\text{cdr}\,(\text{assoc}\,(b,\ s))))\\
&\to\ \ (\text{result}\,(\text{unstack}\,(b),\ s)\\
&\qquad = \ \text{cons}\,(\text{cdr}\,(\text{assoc}\,(b,\ s)),\ \text{cons}\,(\text{list}\,(b),\ \text{delete}\,(\text{assoc}\,(b,\ s),\ s))))
\end{aligned}$$

Because these theorems were also the first thing we proved using the blocks world formalization in Appendix A, we could "load" the sequence of events in Section A.1 without change after loading the events for formalizing the blocks world within the general theory given in Appendix C. That is, the theorem prover verified the same sequence of theorems without any change using the new blocks world theory. In an analogous manner, we could re-define the checkerboard domain as part of our general framework.

5.2 Representing Complex Actions

As observed earlier, complex actions—actions corresponding to plans such as the plan for clearing a block hitherto specified by plan generating programs—can also be represented by lambda expressions that stand for the Lisp functions that give their input-output specification. The function `res-makeclear` given below specifies the input-output behavior of the complex action of clearing a block corresponding to the plan given by `makeclear-gen`. Since we have verified the `makeclear-gen` program, we can use it to express `res-makeclear` as follows.

```
DEFINITION:
makeclear-gen ($b$, $s$)
=  if ($\neg$ find-stack-of-block ($b$, $s$)) $\vee$ ($\neg$ bw-statep ($s$)) then f
   elseif $b$ = car (find-stack-of-block ($b$, $s$)) then nil
   else cons (unstack (car (find-stack-of-block ($b$, $s$))),
              makeclear-gen ($b$,
                             result (unstack (car (find-stack-of-block ($b$,
                                                                       $s$))),
                                     $s$))) endif
```

DEFINITION:
res-makeclear (b, s) = resultlist (makeclear-gen (b, s), s)

In other words, the specification of the complex action is got by "composing" the specifications of the primitive actions. The lambda expression λs. res-makeclear (b, s) can be represented using `make-action` as follows.

DEFINITION:
makeclear (b) = make-action (`'res-makeclear`, list (b))

The following example shows that `result` may be used to do the lambda evaluation even with the complex action `makeclear`.

THEOREM: complex-action-ex1
result (makeclear (`'b`), STATE1) = `'((b c) (a) (d e))`

It should be evident that we can use lambda expressions to represent recursively complex actions corresponding to plans that involve other complex actions. This would help us describe large complex plans succinctly. For instance, a plan to clear a list of blocks using the action `makeclear` can be generated by the following plan generator.

DEFINITION:
makeclear-list-gen (l, s)
= **if** $l \simeq$ **nil then nil**
else cons (makeclear (car (l)),
makeclear-list-gen (cdr (l),
result (makeclear (car (l)), s))) **endif**

This plan clears a list of blocks by clearing them one after the other using `makeclear`. For instance, the plan to clear the list of blocks `'(a b c)` in STATE1 using `makeclear-list-gen` and the state got as a result of executing this plan in STATE1 are shown below.

THEOREM: complex-action-ex2
makeclear-list-gen (`'(a b c)`, STATE1)
=
```
'((res-makeclear 'a)
  (res-makeclear 'b)
  (res-makeclear 'c))
```

THEOREM: complex-action-ex3
resultlist (makeclear-list-gen (`'(a b c)`, STATE1), STATE1)
= `'((c) (b) (a) (d e))`

We can form the complex action corresponding to the plan `makeclear-list-gen` once again using `make-action`. But first we need a specification of this action.

DEFINITION:
res-makeclear-list (l, s) = resultlist (makeclear-list-gen (l, s), s)

DEFINITION:
makeclear-list (l) = make-action (`'res-makeclear-list`, list (l))

The following shows that the result of executing this action in STATE1 is the same as the result of executing the plan generated by the program `makeclear-list-gen`.

THEOREM: complex-action-ex4
result (makeclear-list (`'(a b c)`), STATE1)
= `'((c) (b) (a) (d e))`

5.3 Specifying Partial Actions

Sometimes we may have to make do with a partial specification of actions if the effects of actions on all aspects of the state of the system are not given in advance. Still, it may be possible to solve problems using the given incomplete information. For example, suppose all we know about a blocks world is the following. There are two actions: one to clear any block in any state and another to paint any block with any color in any state. While we are given that painting a clear block leaves the block clear, we are not given whether clearing a block changes its color or not. Suppose the problem is to achieve a state in which a block is both black and clear. A correct solution to the problem is to clear the given block first and then paint it black. We cannot make the assumption that clearing a block does not change its color because it would then allow us to prove that we can paint a block first and then clear it to achieve a state in which a block is both clear and black, a clearly unacceptable solution. Thus, while we can make the assumption that those aspects of the system state specified in a problem describe the state completely, there are circumstances in which we may have to use a partial specification of actions to solve a problem. It is perhaps for this reason that it is sometimes argued that specification languages for programs must not be executable [46]. When actions are partially specified, they cannot be modeled as programs. Therefore, it would not be possible for us to define an interpreter for the system and reason about it as we have been doing so far.

Fortunately, we do not need a different logic for dealing with partial actions. The `CONSTRAIN` event [8] of Nqthm allows us to add axioms that *constrain* function symbols without completely characterizing them. To ensure consistency, we are required to provide an already defined "witness" function that satisfies the proposed axiom about a new function symbol. The defined "witness" function is a model of the new function symbol. In [8], the intuition behind the `CONSTRAIN` event is explained as follows:

> Intuitively, a good way to think about a `CONSTRAIN` event is to imagine defining a new function symbol, proving a theorem about that function symbol, and then forgetting about the defining equation while remembering the theorem.…It turns out that this check guarantees consistency,…

We may specify partial actions using `CONSTRAIN` by first defining a program that completely specifies the action as we have been doing so far and then use it as a "witness" for the partially specified action. We may use (a conjunction) of as many theorems about the complete program as are necessary for characterizing the partial action. To formalize the above blocks world example, we can first define a Lisp program that clears blocks without changing its colors, prove the theorem that it clears blocks and use it to characterize a new function symbol that specifies the partial action. This would enable us to verify the given solution without letting us verify spurious ones.

Of course, actions specified partially cannot be executed on concrete data structures. It so happens that actions specified via `CONSTRAIN` cannot be evaluated by the interpreter `eval$`, i.e., we cannot represent actions by lambda expressions as before and do lambda evaluation using `eval$` to get the appropriate "function call" to the action specification. This is because `eval$` evaluates functions defined as Lisp programs whereas `CONSTRAIN` introduces non-executable function symbols which do not have any "code" associated with them. This means that actions partially specified using `CONSTRAIN` cannot be viewed as part of the "programming language" for plans. One way to get around this problem may be to augment the `CONSTRAIN` event so that a user can declare some of the newly introduced function symbols as *SUBRP*s, primitive function symbols [10, page 134]. If functions specifying partial actions are introduced as *SUBRP*s and *SUBRP* axioms [10, page 134] corresponding to these function symbols are added to the logic by the `CONSTRAIN` event then we could perform lambda evaluation of partially specified actions using `eval$` because `APPLY-SUBR` would then return the desired term. For example, suppose that the function symbol `'foo` that takes one argument (a state argument) is used to specify an action partially using `CONSTRAIN`. Then the *SUBRP* axiom corresponding to `'foo` is

$$(\mathrm{subrp}\,(\texttt{'foo}) = \mathbf{t}) \wedge (\mathrm{apply\text{-}subr}\,(\texttt{'foo},\, l) = \mathrm{foo}\,(\mathrm{car}\,(l)))$$

Chapter 6

Mechanizing Modifications to Domain Specifications

In the previous chapters, we presented a mechanized formal theory for specifying problem domains and verifying plans. While our formal theory is general enough to model a wide range of problem domains, it does not address one other issue in software development viz. *program modification*, the changes that must be made to a program, perhaps many years after it is written, due to changes to its specification. By providing a single mechanized framework as we have done in which various problem domains and proofs of plans can be carried out and in which both actions and their specifications can be composed to form larger units, we have an automated reasoning system that can keep track of previous design histories. This helps reduce the amount of work needed to change a design when changes to specifications are made because as much previous work as possible can be reused. However, even when there are reusable design histories available, the burden of figuring out what to redesign and what to reuse rests with the human designer. Some work on building automated systems that can take into account changes to problem specifications and programs is currently underway [96, 97, 106], particularly in the area of *requirements acquisition* [51].

The idea of constructing a program that can accept as input declarative facts that describe modifications to domain specifications so that the necessary changes to specifications are carried out *automatically* by the program is being pursued by many researchers in artificial intelligence [76, 79, 77, 81]. For this we need a formalism in which not only all possible domains but also facts describing changes to a domain specification can be expressed. Non-monotonic reasoning [38] has been found necessary for dealing with this problem in AI but has proven extremely difficult to formalize [84]. Gelfond and Lifschitz [35] identified a simple class of domains, a class of finite state systems, which they describe using a language called $\mathcal{A}$ for testing various approaches to formalizing non-monotonic reasoning. In this chapter, we show how the problem of domain modification and the needed "non-monotonic" reasoning may be mechanized in the Boyer-Moore logic, a first-order logic, by including domains as terms in the language. We illustrate our approach by formalizing the class of domains that can be specified in $\mathcal{A}$.

6.1 The Problem

The ideas behind the problem of specification modification and the need for non-monotonic reasoning are best illustrated using the class of domains that can be described in $\mathcal{A}$. The domains have the following properties:

1. The states of the system are given by the propositional values of a finite number of boolean variables or *fluents*.
2. Actions are not parameterized and are assumed to be executable in all states. However the actual effect of an action when executed in a state may depend on the fluents that are true in the state.

The language $\mathcal{A}$ is used for describing such domains concisely. Each domain is specified by a set of fluents, a set of actions, a set of propositions that specify the effects of actions (e-propositions) and a set of propositions that specify the values of fluents in various situations (v-propositions).

As an example, consider the following Switch Domain described in $\mathcal{A}$. It consists of fluents *Light1* and *Light2*, an action *Switch1* and the following propositions.

$$\begin{array}{l} \textbf{initially } \neg \textit{Light1}, \\ \textbf{initially } \neg \textit{Light2}, \\ \textit{Switch1} \textbf{ causes } \textit{Light1}. \end{array}$$

The first two lines are v-propositions and the last line is an e-proposition. According to the semantics of $\mathcal{A}$, there are 4 states of the Switch Domain depending on the truth values of *Light1* and *Light2*. There is one action *Switch1* which causes *Light1* to become true when executed in any state. Since the semantics of $\mathcal{A}$ incorporates the default rule known as the "commonsense law of inertia" [61, 36] that an action does not affect a fluent unless otherwise mentioned in a domain description, the execution of *Switch1* in any state is assumed by default to leave the value of *Light2* unchanged. Thus, the Switch Domain specifies a simple finite state machine whose states are given by the boolean values of the fluents *Light*1 and *Light2* at various times and whose transitions are given by the actions of the domain viz. *Switch1*. In addition, the actual values of the fluents in various states starting with an initial state may be specified in a domain description in $\mathcal{A}$ using v-propositions. Roughly speaking, these v-propositions play the role of constraints of Chapter 4. The above domain description says that both *Light1* and *Light2* are false in the initial state.

One form of domain modification that has been frequently considered in formal AI is *addition* of new information to a domain. For instance, we could modify the Switch Domain to the following Extended Switch Domain by adding the following e-proposition:

$$\textit{Switch1} \textbf{ causes } \textit{Light2}.$$

Naturally, this defines a different finite state system and, in general, all facts that are true about the original domain need not remain true about the new domain. For instance, we know that if *Switch1* is executed in a state in which *Light2* is false, then *Light2* will remain false in the Switch Domain but become true in the Extended Switch Domain. The current approach to express domain modifications is by adding formulas that describe a change to a domain to an existing first-order formalization of the domain. However, simply adding formulas describing the new e-proposition to a formalization of the Switch Domain would not give us a formalization of the Extended Switch Domain in classical first-order logic because first-order logic is monotonic: if a formula p can be proved by deduction from a set of formulas Γ then p can also be proved from $\Gamma \cup \{q\}$. Thus, it is felt that a non-monotonic formalism—usually first-order logic augmented by non-monotonic rules such as *circumscription* [77, 80, 61]—is needed to express changes to domain specifications, as the following quotation from Gelfond and Lifschitz [35] indicates.

> The entailment relation of $\mathcal{A}$ is nonmonotonic, in the sense that adding an e-proposition to a domain description D may nonmonotonically change the set of propositions entailed by D. (This cannot happen when a v-proposition is added.) For this reason, a modular translation from $\mathcal{A}$ into another declarative language (that is, a translation that processes propositions one by one) can be reasonably adequate only if this other language is nonmonotonic also.

In a circumscription based formalism similar to [4], (a relevant part of) the Switch Domain may be formalized as follows.

$$\neg noninertial(p, a) \rightarrow (holds(p, s) \equiv holds(p, result(a, s)))$$

$$noninertial(Light1, Switch1)$$

$$holds(Light1, result(Switch1, s))$$

Here variables p, s, and a are universally quantified from the outside over fluents, situations and actions respectively. Circumscription is a rule that is used to minimize the extension of certain predicates common to all domains. By adding facts about these "control" predicates to an existing theory, we can "switch" from a theory of one domain to a theory of another. In the above formalization, the control predicate is *noninertial* which says whether or not an action affects a fluent. The first axiom, sometimes called the "commonsense law of inertia", says that if an action a does not affect a fluent p, then the value of p after a is executed in any situation s

is the same as the value of p in s. We explicitly assert that $Switch1$ affects $Light1$ and specify the effects of $Switch1$ on all fluents that it affects. Since $noninertial$ is minimized via circumscription, $\neg noninertial(Light2, Switch1)$ is provable from the above formulas. Thus, we can prove:

$$holds(Light2, s) \equiv holds(Light2, result(Switch1, s))$$

To obtain the formalization of the Extended Switch Domain without changing the existing axioms, we simply add the following axioms. These clearly invalidate the above inference.

$$noninertial(Light2, Switch1)$$

$$holds(Light2, result(Switch1, s))$$

Thus, we have succeeded in suppressing some of the inferences that are not true in the new domain, i.e., in formalizing non-monotonic reasoning. But as McCarthy [84] explains "the most obvious and apparently natural axiomatizations tend to have unintended models". At present, the formalization of non-monotonic reasoning is a subject of continuing research [84].

6.2 Our Approach

Our approach to mechanize the "non-monotonic" reasoning needed for dealing with modifications to domain specifications in classical first-order logic is to include all possible domains as terms in the logic. This allows us to express directly in first-order logic that "$Switch1$ does not affect $Light2$ when executed in any state of the $SwitchDomain$" and that "$Switch1$ causes $Light2$ to become true when executed in any state in the $ExtendedSwitchDomain$" without the two facts "interfering" with each other and causing a contradiction to be derived. Here $SwitchDomain$ and $ExtendedSwitchDomain$ are terms (data structures) in the logic that represent the Switch Domain and the Extended Switch Domain respectively. This is somewhat similar to the device of introducing explicit state terms that allowed us to express in first-order logic facts that may be true in one situation and false in another without them interfering with each other and resulting in a contradiction. Previously we had only formalized all possible states and all possible actions of all possible domains but we did not explicitly represent the information regarding either the possible domains or the information about which states and actions belonged to a domain. This information was available to the human user who used it to manually construct the formalization of any given domain from the available formalization of the states and actions of the domain. By including all possible domains as terms in the logic, we can give a program the knowledge about the possible domains and the information about what happens when an action is executed in a particular state of a particular domain. Thus, the inclusion of domains as terms in the logic amounts to formalizing the meta-theory used by designers of non-monotonic rules to prove their soundness and completeness.

Once all possible domains are available as terms, we want to express formally the information about what states and actions belong to a domain and what happens when an action is executed in a state of a domain. As before, the set of states of a domain and the set of actions of a domain can be formalized by writing suitable Lisp predicates. To express theorems about the effects of executing an action in a state of a domain, we define the functions *result* and *resultlist* with an extra domain parameter. So we have terms such as result (`'switch1`, s, SWITCH-DOM2) which stands for the state got as a result of executing $Switch1$ in state s of the Switch Domain. We may then prove theorems such as the following:

THEOREM: l-th3
(switch-dom2-statep (s) $\wedge$ holds (`'(light2 . 0)`, s))
$\rightarrow$ holds (`'(light2 . 1)`, result (`'switch1`, s, SWITCH-DOM2))

The above theorem says that if $Light2$ (represented by `'light2`) is false in any state of the Extended Switch Domain (SWITCH-DOM2) then it will become true when $Switch1$ (`'switch1`) is executed.

Observe that our approach is independent of whether the domains are described in $\mathcal{A}$ or in some other language; all we need is a representation of domains in Lisp that would allow us to define the *result* function. We must choose our representation so that small changes to a domain specification are expressed fairly easily in the logic. We therefore represent domains by the list of e-propositions contained in their description in $\mathcal{A}$. Thus, SWITCH-DOM2 in the theorem l-th2 given above is a list of e-propositions (represented in Lisp) of the Extended Switch Domain. To compute the next state produced by the execution of an action a in a state s of domain D, *result* goes through the list of effect propositions in D, obtains a list of e-propositions that mention a, and computes the next state based on the effects of a specified by those e-propositions. Thus, the "commonsense law of inertia" is expressed in Lisp as part of the definition of *result*. As mentioned in Chapter 1, this approach has a number of advantages. In particular, we can now define plan generating Lisp programs that generate distinct solutions to a problem depending on the domain passed to it as a parameter. Such a program essentially provides an automatic solution to the problem of program modification: when the domain specification is changed, the solution corresponding to the new domain can be immediately generated by passing the new domain as a parameter to the plan generating program.

The rest of the chapter is organized as follows. In Section 6.3, we reproduce almost verbatim the rigorous definition of the syntax and semantics of $\mathcal{A}$ along with the examples in [35]. These examples have been used by researchers in commonsense reasoning as tests or counterexamples to non-monotonic formalisms at various times. In Section 6.4, we describe our formalization of the class of domains that can be specified in $\mathcal{A}$. In Section 6.5, we formalize the examples given in 6.3 and state theorems about the effects of executing sequences of actions in various domains. We show how the needed "non-monotonic" reasoning is formalized in our theory using the Switch Domain example described above. Section 6.6 deals with a class of theorems motivated by [104] about properties true of all states of a domain reachable from an initial state. Section 6.7 describes a plan generating program that outputs distinct solutions depending on the domain passed to it as a parameter. The mechanical proofs of the theorems mentioned in this chapter were carried out *automatically* by the theorem prover without any assistance from the user. The list of events given to the theorem prover for mechanizing $\mathcal{A}$ is in Appendix D. The theorems proved about the examples in Section 6.3 and the events for verifying the plan generating program in Section 6.7 are given in Sections D.1 and D.2 of Appendix D.

6.3 The Language $\mathcal{A}$

A description of an action domain in the language $\mathcal{A}$ consists of "propositions" of two kinds. A "v-proposition" specifies the value of a fluent in a particular situation—either in the initial situation, or after performing a sequence of actions. An "e-proposition" describes the effect of an action on a fluent.

We begin with two disjoint nonempty sets of symbols, called *fluent names* and *action names*. A *fluent expression* is a fluent name possibly preceded by $\neg$. A *v-proposition* is an expression of the form

$$F \textbf{ after } A_1; \ldots; A_m, \tag{6.1}$$

where F is a fluent expression, and $A_1,\ldots,A_m$ $(m \geq 0)$ are action names. If $m = 0$, we will write (6.1) as

$$\textbf{initially } F.$$

An *e-proposition* is an expression of the form

$$A \textbf{ causes } F \textbf{ if } P_1, \ldots, P_n, \tag{6.2}$$

where A is an action name, and each of $F, P_1, \ldots, P_n$ $(n \geq 0)$ is a fluent expression. About this proposition we say that it *describes the effect of A on F*, and that $P_1, \ldots, P_n$ are its *preconditions*. If $n = 0$, we will drop **if** and write simply

$$A \textbf{ causes } F.$$

A *proposition* is a v-proposition or an e-proposition. A *domain description*, or simply *domain*, is a set of propositions (not necessarily finite). Lifschitz (private communication) adds that "strictly speaking, a domain description includes, in addition to its value and effect propositions, the lists of its fluent names and of its action names."

Example 1. The Fragile Object domain, motivated by an example from [108], has the fluent names *Holding*, *Fragile* and *Broken*, and the action *Drop*. It consists of two e-propositions:

$$\begin{aligned}&\mathit{Drop}\ \textbf{causes}\ \neg \mathit{Holding}\ \textbf{if}\ \mathit{Holding},\\ &\mathit{Drop}\ \textbf{causes}\ \mathit{Broken}\ \textbf{if}\ \mathit{Holding}, \mathit{Fragile}.\end{aligned}$$

Example 2. The Yale Shooting domain, motivated by the example from [43], is defined as follows. The fluent names are *Loaded* and *Alive*; the action names are *Load*, *Shoot* and *Wait*. The domain is characterized by the propositions

$$\begin{aligned}&\textbf{initially}\ \neg \mathit{Loaded},\\ &\textbf{initially}\ \mathit{Alive},\\ &\mathit{Load}\ \textbf{causes}\ \mathit{Loaded},\\ &\mathit{Shoot}\ \textbf{causes}\ \neg \mathit{Alive}\ \textbf{if}\ \mathit{Loaded},\\ &\mathit{Shoot}\ \textbf{causes}\ \neg \mathit{Loaded}.\end{aligned}$$

Example 3. The Murder Mystery domain, motivated by an example from [4], is obtained from the Yale Shooting domain by substituting

$$\neg \mathit{Alive}\ \textbf{after}\ \mathit{Shoot}; \mathit{Wait} \tag{6.3}$$

for the proposition **initially** $\neg$*Loaded*.

Example 4. The Stolen Car domain, motivated by an example from [55], has one fluent name *Stolen* and one action name *Wait*, and is characterized by two propositions:

$$\begin{aligned}&\textbf{initially}\ \neg \mathit{Stolen},\\ &\mathit{Stolen}\ \textbf{after}\ \mathit{Wait}; \mathit{Wait}; \mathit{Wait}.\end{aligned}$$

The Stolen Car domain shows that inconsistent constraints may be specified about a domain. An adequate formalization must allow us to prove that the given information is inconsistent whenever possible.

To describe the semantics of $\mathcal{A}$, we will define what the "models" of a domain description are, and when a v-proposition is "entailed" by a domain description.

A *state* is a set of fluent names. Given a fluent name F and a state σ, we say that F *holds* in σ if $F \in \sigma$; $\neg F$ *holds* in σ if $F \notin \sigma$. A *transition function* is a mapping Φ of the set of pairs (A, σ), where A is an action name and σ is a state, into the set of states. A *structure* is a pair (σ_0, Φ), where σ_0 is a state (the *initial state* of the structure), and Φ is a transition function.

For any structure M and any action names $A_1, \ldots, A_m$, by $M^{A_1;\ldots;A_m}$ we denote the state

$$\Phi(A_m, \Phi(A_{m-1}, \ldots, \Phi(A_1, \sigma_0) \ldots)),$$

where Φ is the transition function of M, and σ_0 is the initial state of M. We say that a v-proposition of the form given by (6.1) above is *true* in a structure M if F holds in the state $M^{A_1;\ldots;A_m}$, and that it is *false* otherwise. In particular, a proposition of the form **initially** F is true in M iff F holds in the initial state of M.

A structure (σ_0, Φ) is a *model* of a domain description D if every v-proposition from D is true in (σ_0, Φ), and, for every action name A, every fluent name F, and every state σ, the following conditions are satisfied:

(i) if D includes an e-proposition describing the effect of A on F whose preconditions hold in σ, then $F \in \Phi(A, \sigma)$;

(ii) if D includes an e-proposition describing the effect of A on $\neg F$ whose preconditions hold in σ, then $F \notin \Phi(A, \sigma)$;

(iii) if D does not include such e-propositions, then $F \in \Phi(A, \sigma)$ iff $F \in \sigma$.

It is clear that there can be at most one transition function Φ satisfying conditions (i)–(iii). Consequently, different models of the same domain description can differ only by their initial states. For instance, the Fragile Object domain (Example 1) has 8 models, whose initial states are the subsets of

$$\{Holding, Fragile, Broken\};$$

in each model, the transition function is defined by the equation

$$\Phi(Drop, \sigma) = \begin{cases} \sigma \setminus \{Holding\} \cup \{Broken\}, & \text{if } Holding, Fragile \in \sigma, \\ \sigma \setminus \{Holding\}, & \text{otherwise.} \end{cases}$$

A domain description is *consistent* if it has a model, and *complete* if it has exactly one model. The Fragile Object domain is consistent, but incomplete. The Yale Shooting domain (Example 2) is complete; its only model is defined by the equations

$$\begin{aligned} &\sigma_0 = \{Alive\}, \\ &\Phi(Load, \sigma) = \sigma \cup \{Loaded\}, \\ &\Phi(Shoot, \sigma) = \begin{cases} \sigma \setminus \{Loaded, Alive\}, & \text{if } Loaded \in \sigma, \\ \sigma, & \text{otherwise,} \end{cases} \\ &\Phi(Wait, \sigma) = \sigma. \end{aligned}$$

The Murder Mystery domain (Example 3) is complete also; it has the same transition function as Yale Shooting, and the initial state $\{Loaded, Alive\}$. The Stolen Car domain (Example 4) is inconsistent.

A v-proposition is *entailed* by a domain description D if it is true in every model of D. For instance, Yale Shooting entails

$$\neg Alive \textbf{ after } Load; Wait; Shoot.$$

Murder Mystery entails, among others, the propositions

$$\textbf{initially } Loaded$$

and

$$\neg Alive \textbf{ after } Wait; Shoot.$$

Note that the last proposition differs from (6.3) by the order in which the two actions are executed. This example illustrates the possibility of reasoning about alternative "possible futures" of the initial situation.

6.4 Our Formalization

We restrict ourselves to the class of domains[1] that can be described in $\mathcal{A}$ by a finite number of propositions. To formalize the semantics of $\mathcal{A}$, we need to represent all possible states of all domains, all possible sequences of actions of all domains and all possible domains, i.e., finite sets of e-propositions as terms in the logic. We represent fluents and actions as litatoms.

DEFINITION: $\text{fluentp}\,(x) = \text{litatom}\,(x)$

DEFINITION: $\text{actionp}\,(x) = \text{litatom}\,(x)$

A state is represented as a list of pairs of the form $cons(fluent, value)$ where $value$ is either 0 or 1. A fluent is true or false in a state depending on whether it is paired with 1 or 0. The predicate `valuep` checks if its argument is either 0 or 1, the predicate `fvlistp` checks if its argument is a list of fluent-value pairs and `a-statep` is a predicate on all possible states of all possible domains.

DEFINITION: $\text{valuep}\,(x) = ((x = \texttt{0}) \lor (x = \texttt{1}))$

[1] A clarification about terminology. We will use the word "domain" to refer to problem domains as we have been doing so far in this dissertation. When we want to refer to a description of a domain in $\mathcal{A}$, we will explicitly say so. In other words, domains are semantic entities which may be represented in many ways one of which is as a domain description in $\mathcal{A}$.

Definition:

```
fvlistp $(s)$
= if $s \simeq$ nil then t
  else fluentp (caar $(s)$)
       $\wedge$ valuep (cdar $(s)$)
       $\wedge$ fvlistp (cdr $(s)$) endif
```

Definition: a-statep (x) = fvlistp (x)

While `a-statep` can be used to prove general properties about states of all domains, the following predicate `d-statep` may be used to define a predicate on the states on any particular domain. It uses the predicate `has-fluents` to check if all the fluents in the given fluent list l occur in given state s.

Definition:

```
has-fluents $(l, s)$
= if $l \simeq$ nil then t
  else assoc (car $(l)$, $s$) $\wedge$ has-fluents (cdr $(l)$, $s$) endif
```

Definition:
d-statep (l, s) = (a-statep (s) $\wedge$ has-fluents (l, s))

Here is an example of a state in the Fragile Object domain in which the fluents *Holding* and *Fragile* are true and *Broken* is false.

Definition:
STATE1 = `'((holding . 1) (fragile . 1) (broken . 0))`

Theorem: state-example1
a-statep (STATE1)

We allow more than one pair with the same car in a state but take into account only the first pair in the list in which a fluent occurs when evaluating the truth value of the fluent. This representation of states simplifies the formalization and makes it easier to carry out mechanical proofs. Below we give another state STATE2 that is equivalent to STATE1 defined above. The fluent `'broken` is false in STATE2 because we only consider the value associated with the first pair in the list whose car is `'broken`.

DEFINITION:
STATE2
$=$ `'((holding . 1)`
`(fragile . 1)`
`(broken . 0)`
`(broken . 1))`

THEOREM: state-example2
a-statep (STATE2)

The predicate `frag-statep` defined below recognizes the states of the Fragile Object Domain.

DEFINITION:
frag-statep (s) = d-statep (`'(holding broken fragile)`, s)

THEOREM: state-example3
frag-statep (STATE1) $\wedge$ frag-statep (STATE2)

We define a predicate `holds` that takes a fluent expression (a fluent-value pair) and a state as arguments and checks if the fluent expression is true in the state. Since we allow a fluent to occur as the car of more than one fluent-value pair in a state, `holds` checks the first pair in the given state whose car is equal to the car of the given fluent expression using the built-in function `assoc`. Thus, $\neg Broken$ holds in both STATE1 and STATE2 defined earlier.

DEFINITION: holds (*fexp*, s) = (assoc (car (*fexp*), s) = *fexp*)

THEOREM: holds-example1
holds (`'(broken . 0)`, STATE1)

THEOREM: holds-example2
holds (`'(broken . 0)`, STATE2)

`Holds-all` checks if all the fluent expressions in the given list x hold in state s.

DEFINITION:
holds-all (x, s)
$=$ **if** $x \simeq$ **nil then t**
else (assoc (caar (x), s) = car (x)) $\wedge$ holds-all (cdr (x), s) **endif**

Domains are represented as lists of e-propositions that occur in their description in $\mathcal{A}$. We represent an e-proposition as a list whose car is an action, cadr is an effect which is a fluent expression and the rest are fluent expressions corresponding to *preconditions*. Thus, an e-proposition is a list whose car is an action and whose cdr is a list of fluent-value pairs.

DEFINITION:
e-prop (x) = (actionp (car (x)) $\wedge$ fvlistp (cdr (x)))

For instance, the e-proposition:

Drop **causes** *Broken* **if** *Holding*, *Fragile*.

is represented as shown in the theorem below.

THEOREM: eprop-example1
e-prop (`'(drop (broken . 1) (holding . 1) (fragile . 1))`)

The predicate `domainp` recognizes a list of e-propositions.

DEFINITION:
domainp (x)
= **if** $x \simeq$ **nil then t**
  else e-prop (car (x)) $\wedge$ domainp (cdr (x)) **endif**

The example below shows a representation of the Fragile Object domain.

DEFINITION:
FO-DOMAIN

```
=  '((drop (holding . 0) (holding . 1))
     (drop (broken . 1) (holding . 1) (fragile . 1)))
```

THEOREM: domainp-example1
domainp (FO-DOMAIN)

The interpreter `resultlist` takes a list of actions l, a state s and a domain *dom* as arguments and returns the state got by executing l in s according to the transition function specified by *dom*. The interpretation of effect propositions is done by the function `compute-effects`. `Compute-effects` returns a list containing all fluent-value pairs made true by executing an action a in the given state s according to the given set of effect propositions *dom*. The "common sense law of inertia" is expressed using this function. The function goes through the list of effect propositions given by *dom* and returns a list of all fluent expressions P such that there exists an effect proposition e describing the effect of a on P in *dom* and the preconditions specified in e hold in s.

DEFINITION:
```
compute-effects ($a$, $s$, $dom$)
=  if $dom \simeq$ nil then nil
   elseif ($a$ = caar ($dom$)) $\wedge$ holds-all (cddar ($dom$), $s$)
   then cons (cadar ($dom$), compute-effects ($a$, $s$, cdr ($dom$)))
   else compute-effects ($a$, $s$, cdr ($dom$)) endif
```

Here are some theorems that show how `compute-effects` operates. The list of effects of executing `'drop` in STATE1 includes $\neg Holding$ and $Broken$ since the preconditions of both the effect propositions of the Fragile Object Domain hold in STATE1.

THEOREM: compute-effects1
compute-effects (`'drop`, STATE1, FO-DOMAIN)
= `'((holding . 0) (broken . 1))`

However, the effect of `'drop` in STATE3 defined below is just $\neg Holding$.

DEFINITION:
STATE3 = `'((holding . 1) (broken . 0) (fragile . 0))`

THEOREM: compute-effects2
compute-effects (`'drop`, STATE3, FO-DOMAIN) = `'((holding . 0))`

The function `result` returns the state got by executing an action a in s according to the transition function specified by dom. The effects of the action in s computed by `compute-effects` are appended in front of s because we only look at the first occurrence of a fluent in a state to determine its truth value.

DEFINITION:
result (a, s, dom) = append (compute-effects (a, s, dom), s)

THEOREM: result-example1
result (`'drop`, STATE1, FO-DOMAIN)
```
=  '((holding . 0)
     (broken . 1)
     (holding . 1)
     (fragile . 1)
     (broken . 0))
```

THEOREM: result-example2
result (`'drop`, STATE3, FO-DOMAIN)
=
```
'((holding . 0)
  (holding . 1)
  (broken . 0)
  (fragile . 0))
```

Notice that we do not allow an inconsistent set of effect propositions to occur in a domain. If a list of e-propositions that contained both

$$A \textbf{ causes } P$$

and

$$A \textbf{ causes } \neg P$$

is given as an argument to *result*, then the execution of the action will make either P or $\neg P$ true depending on the order in which the propositions appear in the list. This is not a limitation because we are only interested in representing the class of possible domains and there is no domain corresponding to an inconsistent set of propositions. Recall that we are only using the syntax of $\mathcal{A}$ to represent domains as terms in the logic.

The interpreter *resultlist* and an example of its output on concrete data are given below. When `'drop` is executed in STATE1 it has the effect of making *Holding* false and *Broken* true. But when it is executed again it has no effect since the preconditions of none of the effect propositions in the domain are true in the new state.

DEFINITION:
resultlist (l, s, dom)
= **if** $l \simeq$ **nil then** s
else resultlist (cdr (l), result (car (l), s, dom), dom) **endif**

THEOREM: resultlist-example1
resultlist (`'(drop drop)`, STATE1, FO-DOMAIN)
=
```
'((holding . 0)
  (broken . 1)
  (holding . 1)
  (fragile . 1)
  (broken . 0))
```

6.5 Examples

We are now ready to formalize the example domains given in Section 6.3 and prove whether or not a sequence of actions brings about a goal state when executed in an initial state of a domain. The theorems given below were proved completely automatically by the theorem prover. These theorems are distinctly simpler than the theorems shown in the previous chapters because they have been used so far for testing non-monotonic formalisms.

From the description of the Fragile Object Domain, the following theorems similar to those given in [108] about the effects of the drop action follow. Below we state the theorem that if *Holding*, *Fragile* and $\neg$*Broken* are true in state s in the Fragile Object Domain, then executing *Drop* in s results in a state in which $\neg$*Holding*, *Fragile* and *Broken* are true.

DEFINITION: holding (s) = holds (`'(holding . 1)`, s)

DEFINITION: fragile (s) = holds (`'(fragile . 1)`, s)

DEFINITION: broken (s) = holds (`'(broken . 1)`, s)

DEFINITION:
frag-statep (s) = d-statep (`'(holding broken fragile)`, s)

THEOREM: frag-th1
(frag-statep ($s0$)
 $\wedge$ holding ($s0$)
 $\wedge$ fragile ($s0$)
 $\wedge$ ($\neg$ broken ($s0$))
 $\wedge$ ($s1$ = result (`'drop`, $s0$, FO-DOMAIN)))
 $\rightarrow$ (($\neg$ holding ($s1$)) $\wedge$ fragile ($s1$) $\wedge$ broken ($s1$))

Here is another example from the Fragile Object Domain. If *Holding* and $\neg$*Broken* are true in $s0$ of the Fragile Object Domain then $\neg$*Holding* will be true on executing *Drop* in $s0$.

THEOREM: frag-th2
(frag-statep ($s0$) $\wedge$ holding ($s0$) $\wedge$ ($\neg$ broken ($s0$)))
 $\rightarrow$ ($\neg$ holding (result (`'drop`, $s0$, FO-DOMAIN)))

The set of effect propositions in the Yale Shooting Domain is defined below and is followed by the usual theorem about the death of Fred.

DEFINITION:
YALE-DOMAIN
= `'((load (loaded . 1))`
`(shoot (alive . 0) (loaded . 1))`
`(shoot (loaded . 0)))`

The following theorem says that if $\neg Loaded$ and *Alive* hold in $s0$ then $\neg Alive$ will be true in the state got by executing the sequence of actions, `'(load wait shoot)`, in $s0$.

DEFINITION:
yale-statep (s) = d-statep (`'(alive loaded)`, s)

THEOREM: ysp1
(yale-statep (*s0*)
∧ holds (`'(loaded . 0)`, *s0*)
∧ holds (`'(alive . 1)`, *s0*))
→ holds (`'(alive . 0)`,
resultlist (`'(load wait shoot)`, *s0*, YALE-DOMAIN))

The following theorem says that shooting will unload the gun in any state of the Yale Shooting Domain.

THEOREM: ysp2
yale-statep (*s0*)
→ (¬ holds (`'(loaded . 1)`, result (`'shoot`, *s0*, YALE-DOMAIN)))

The Murder Mystery example requires reasoning about the past. The problem is to deduce that the gun was loaded in the initial state given that Fred was alive in the initial state and not alive after *Shoot* and *Wait*.

THEOREM: mm1
(yale-statep (*s0*)
∧ holds (`'(alive . 1)`, *s0*)
∧ holds (`'(alive . 0)`,
resultlist (`'(shoot wait)`, *s0*, YALE-DOMAIN)))
→ holds (`'(loaded . 1)`, *s0*)

The v-proposition

$$\neg Alive \textbf{ after } Wait; Shoot.$$

is stated as follows.

THEOREM: mm2
(yale-statep ($s0$)
$\wedge$ holds (`'(alive . 1)`, $s0$)
$\wedge$ holds (`'(alive . 0)`,
resultlist (`'(shoot wait)`, $s0$, YALE-DOMAIN)))
$\rightarrow$ holds (`'(alive . 0)`,
resultlist (`'(wait shoot)`, $s0$, YALE-DOMAIN))

The Stolen Car domain description in Section 6.3. gives inconsistent information because it is specified that a car not stolen in the initial state gets stolen after *Wait* is executed thrice in succession. The following theorem says that the Stolen Car Domain is inconsistent.

DEFINITION: stolen (s) = holds (`'(stolen . 1)`, s)

THEOREM: st2
(($\neg$ stolen ($s0$))
$\wedge$ stolen (resultlist (`'(wait wait wait)`, $s0$, **nil**)))
$\rightarrow$ **f**

We now show examples that involve modifications to domains. Our first modification does not lead to "non-monotonic" reasoning. Below we show that we can add the e-proposition

Shoot **causes** $\neg$*Alive* **if** *VeryNervous*.

to the Yale Shooting Domain and prove the theorem that if the victim is very nervous in the initial situation then he will die after shoot occurs.

DEFINITION:
YALE-DOMAIN1
```
= '((load (loaded . 1))
    (shoot (alive . 0) (loaded . 1))
    (shoot (loaded . 0))
    (shoot (alive . 0) (verynervous . 1)))
```

THEOREM: ysp3
holds (`'(verynervous . 1)`, s)
$\rightarrow$ holds (`'(alive . 0)`, result (`'shoot`, s, YALE-DOMAIN1))

As described earlier, even when a modification to a domain results in "non-monotonic" reasoning, it can be dealt with easily in our theory. This is because all possible domains are represented as terms in the logic. Thus, theorems about various domains can co-exist in the theory without contradicting each other. Moreover, our representation of the domains, allows us to describe modifications to domains conveniently. The theorem that *Light2* remains false after *Switch1* is executed in the Switch Domain (SWITCH-DOM1) is stated as follows.

DEFINITION:
SWITCH-DOM1 = `'((switch1 (light1 . 1)))`

THEOREM: l-th1
holds (`'(light2 . 0)`, s)
→ holds (`'(light2 . 0)`, result (`'switch1`, s, SWITCH-DOM1))

If the Switch Domain is extended with the e-proposition:

Switch1 **causes** *Light2*

the theorem that *Switch*1 will make *Light*2 true can be proved about this Extended Switch Domain named SWITCH-DOM2.

DEFINITION:
SWITCH-DOM2
= `'((switch1 (light1 . 1)) (switch1 (light2 . 1)))`

THEOREM: l-th2
holds (`'(light2 . 0)`, s)
→ holds (`'(light2 . 1)`, result (`'switch1`, s, SWITCH-DOM2))

We can also express the above theorem without explicitly constructing the Extended Switch Domain. That is, we can say that "*Switch1* makes *Light2* true in the domain got by adding the new e-proposition to the Switch Domain".

THEOREM: l-th4
holds (`'(light2 . 0)`, s)
→ holds (`'(light2 . 1)`,
result (`'switch1`,
s,
cons (`'(switch1 (light2 . 1))`, SWITCH-DOM1)))

6.6 Proving Properties of States

Reiter [104] motivates the need to prove that certain properties are true in all states of a domain accessible from the initial state. Such theorems are typical of many domains. In fact, we have already proved a fair number of them in the preceding chapters. Below we prove a few typical examples similar to those discussed in the literature. As we have seen before, the proofs of such properties require mathematical induction. Lifschitz (private communication) observed that most of these theorems assumed the following general form: Executing any sequence of actions whose elements belong to V (where V is some set of actions) in the initial situation leads to a situation in which F holds (where F is some fluent expression). Below we define two predicates `klp` and `always-holds` and show how this class of theorems can be expressed in our formal theory. When the actions in V are exactly the actions of a domain, then this amounts to proving that F is true in all states reachable from the initial state of the domain.

The predicate `klp` ("Kleene predicate") checks if a given plan p consists only of actions in the set *aset*, i.e., $p \in aset^*$. (Here $*$ is the Kleene $*$ used to construct the set of all sequences of symbols in the action set).

DEFINITION:
klp (p, *aset*)
$=$ **if** $p \simeq$ **nil then** $p =$ **nil**
  else (car (p) $\in$ *aset*) $\wedge$ klp (cdr (p), *aset*) **endif**

The predicate `always-holds` checks if the fluent expression *fexp* is true in the state got as a result of executing the plan p in $s1$ whenever $p \in aset^*$. The semantics of actions are governed by the set of effect propositions *dom* passed as a parameter.

DEFINITION:
always-holds (*fexp*, p, *aset*, *s1*, *dom*)
$=$ (klp (p, *aset*) $\rightarrow$ holds (*fexp*, resultlist (p, *s1*, *dom*)))

Here are some examples. The following theorem says that if Fred is dead in a situation then he cannot be revived by any action in the Yale Shooting Domain.

THEOREM: fred-dead2
holds (`'(alive . 0)`, *s1*)
$\rightarrow$ always-holds (`'(alive . 0)`,
  p,
  `'(load wait shoot)`,
  s1,
  YALE-DOMAIN)

If an object is broken in a situation then it cannot be repaired using the actions of the Fragile Object Domain.

THEOREM: always-broken
holds (`'(broken . 1)`, s)
$\rightarrow$ always-holds (`'(broken . 1)`, p, `'(drop)`, s, FO-DOMAIN)

McCarthy [82] describes how non-monotonic reasoning might be required if a system must take into account a previously unknown action to infer that a previously unachievable goal can now be achieved using the new action. A similar circumstance can be recreated if we add the following e-proposition to the Fragile Object Domain.

Repair **causes** $\neg$*Broken*

With the additional action *Repair* it can be shown that a broken object can be repaired whereas previously it couldn't be. The revised Fragile Object Domain and the theorem that a broken object can be repaired in any situation using the new information are given below.

DEFINITION:
FO-DOMAIN1
```
= '((drop (holding . 0) (holding . 1))
    (drop (broken . 1) (holding . 1) (fragile . 1))
    (repair (broken . 0)))
```

THEOREM: try1
holds (`'(broken . 1)`, s)
$\rightarrow$ holds (`'(broken . 0)`, result (`'repair`, s, FO-DOMAIN1))

6.7 Verifying Plans Common to a Class of Domains

In this section, we show how we can express and verify solutions to problems common to a class of domains using our formalization. As before, we express plans by writing plan generating programs and proving them correct. Because we have a representation in Lisp of all possible domains under consideration, a plan generating program may take an extra domain parameter and generate plans depending on the domain passed to it to solve problems common to a class of domains. Such a program *automatically* takes into account changes to a plan needed as a result of changes to a domain specification. When a domain is changed, we simply need to pass the new domain as the argument to the plan generating program instead of the old to get the new solution. We can prove that the plan generating program works for all *domains* in the class by quantifying over domains.

The plan generating program we have chosen as our example is a general purpose "planning" program for the class of domains that can be described in $\mathcal{A}$. The problem (or class of problems) it solves is the problem of transforming any initial state to any goal state. This problem is common to all domains. However, a problem given by initial condition and goal may be solvable in some domains and not in others. The planner accepts an initial state s, a domain, a goal (a list of fluent expressions) and a number n indicating the length of the plan needed to solve the problem of transforming s into a state that satisfies the goal as input, and outputs either a sequence of actions of length n that can do the job or **f** if no such plan exists. Our "planner" goes through the given domain and constructs a list of actions that occur in the domain. It then constructs a list of all possible sequences of those actions of length n and checks if any of the members of the list solve the given problem. If so, it returns the first such solution, otherwise it returns **f**. The Lisp program is a straightforward breadth-first planner that generates solutions to problems depending on the domain passed to it as an argument and by itself is uninteresting. In fact, a major portion of research in planning in AI is concerned with the construction of more efficient planners of this sort [113, 2]. However, because it is a function in a logic and because we have all possible domains as terms in the logic, it can be proved correct once and used to generate solutions to problems when a domain is modified. Below we state the theorem that the breadth first planner can be used to solve any problem in any domain whenever the problem has a solution.

The function `collect-actions` gathers a list of all actions *actlist* occurring in the e-propositions of the given domain *dom*.

```
DEFINITION:
collect-actions (dom)
=  if dom ≃ nil then nil
   else cons (caar (dom), collect-actions (cdr (dom))) endif
```

The list of all possible sequences of length n of the actions in *actlist* returned by `collect-actions` is constructed by the function `form-all-plans`. `Form-all-plans` uses two auxiliary functions. `Add-action-to-plan` takes a plan p and *actlist* as input and returns a list of plans got by consing each member of *actlist* onto p.

```
DEFINITION:
add-action-to-plan (p, actlist)
=  if actlist ≃ nil then nil
   else cons (cons (car (actlist), p),
                  add-action-to-plan (p, cdr (actlist))) endif
```

`Form-longer-plans` takes a list of plans of length k, *plist*, and a list of actions *actlist* as input and returns a list of plans of length $k + 1$ got by consing every member of *actlist* onto every member of *plist* using `add-action-to-plan`.

DEFINITION:
```
form-longer-plans ($plist$, $actlist$)
=  if $plist \simeq$ nil then nil
   else append (add-action-to-plan (car ($plist$), $actlist$),
                form-longer-plans (cdr ($plist$), $actlist$)) endif
```

`Form-all-plans` is defined as follows.

DEFINITION:
```
form-all-plans ($actlist$, $n$)
=  if $n \simeq$ 0 then list (nil)
   else form-longer-plans (form-all-plans ($actlist$, $n - 1$),
                           $actlist$) endif
```

Once all possible plans using the actions of a domain are available, we must test them and return a solution if one is available. `Get-plan` does this.

DEFINITION:
```
get-plan ($plist$, $dom$, $goal$, $s$)
=  if $plist \simeq$ nil then f
   elseif holds-all ($goal$, resultlist (car ($plist$), $s$, $dom$))
   then car ($plist$)
   else get-plan (cdr ($plist$), $dom$, $goal$, $s$) endif
```

`Find-plan` is the brute-force planner that would generate a plan of length n to solve a problem depending on the domain passed to it.

DEFINITION:
```
find-plan ($s$, $dom$, $goal$, $n$)
=  get-plan (form-all-plans (collect-actions ($dom$), $n$), $dom$, $goal$, $s$)
```

The following theorem says that if there is a plan p in domain x to transform an initial state $s0$ to a state that satisfies the goal $goal$, then find-plan (x, $goal$, len (p)) is one such plan.

THEOREM: planner-complete
(domainp (x)
 $\wedge$ klp (p, collect-actions (x))
 $\wedge$ a-statep ($s0$)
 $\wedge$ fvlistp ($goal$)
 $\wedge$ holds-all ($goal$, resultlist (p, $s0$, x)))
 $\rightarrow$ holds-all ($goal$, resultlist (find-plan ($s0$, x, $goal$, len (p)), $s0$, x))

Chapter 7

Conclusion

The main contribution of this dissertation is a single mechanized theory for specifying problem domains and verifying program-like plans, a single programming language-like representation of plans common to all domains and a method of modeling domains conveniently within our framework. A problem domain specifies the set of possible states of a physical system and a set of actions that can be executed sequentially to change state. A problem specifies, in addition, an initial condition and a goal condition. A solution is a plan which resembles an imperative program for transforming any initial state of the domain to a goal state. A number of systems ranging from conventional von Neumann machines to robots can be viewed as problem domains. The main application we envisage for the mechanized theory presented in this dissertation is software development, be it software for robots or sequential, imperative programs to be run on computers. At various stages of the software life cycle, a program design is expressed as a set of primitive actions and a strategy to combine them, using constructs for conditional execution, iteration and procedure composition, into a program that satisfies a given problem specification. At present, there is considerable emphasis on verifying that a program meets its specification as early as possible in the software life cycle preferably well before the program is coded in a specific programming language. Our mechanized theory is suitable for expressing program designs as plans at various stages of software development and for verifying whether or not such a plan solves the problem under consideration. Our emphasis in this dissertation, though, has been on verifying plans with respect to the requirements specification, the specification of programs in terms of objects of the physical world rather in terms of data structures of a programming language.

There are a number of advantages to our approach. Our method of specifying problem domains and verifying plans does not suffer from many of the limitations of current approaches such as the need for stating explicitly a large number of separate frame axioms and state constraints necessary for reasoning about actions with side-effects. Our plan representation has the expressive power of a programming language in that we can express solutions that involve sequencing, conditional execution, iteration and composition of actions. Our formalization also allows us to verify mechanically many other properties of plans such as efficiency requirements. Because both specifications and plans are executable they can be tested a la programs.

In addition, the dissertation addresses the problem of program modification, the problem of making changes to a plan efficiently when changes are made to the domain specification. Because we have a single mechanized framework for developing programs, we can minimize the work required for redesigning and proving a plan correct when a domain specification is modified by making use of as much of the previous development effort (subplans, specifications, proofs, etc.) as possible. Our approach to dealing with incremental changes to domain specifications is to represent domains as objects in the logic so that solutions to problems common to a class of domains can be expressed and proved correct. Since such solutions are parameterized by domains, the solution to a changed domain can be produced automatically. We show that by choosing a suitable representation for domains, small modifications to domains can be expressed in the logic itself.

7.1 Future Work

There are many directions in which the work described here can be extended. One route is to apply the theory to the task of mechanizing larger, more realistic problem domains than those given in the dissertation. Another is to carry out a complete top-down, stepwise development of a sizable program. A third is to try and obtain similar mechanizations of more complicated classes of problem domains such as those in which the actions of the environment must be taken into account or in which concurrent actions are allowed. Yet another thing that we would like to do is to explore applications where partial actions may be needed and figure out how `eval$` can be made to work on functions introduced via `CONSTRAIN`. We hope that the mechanized theory presented here would be useful for tackling these problems.

Appendix A

Formalization of the Blocks World

This file describes a formalization of the blocks world described in Chapter 1 using our method of modeling domains. There are two actions: move a block to the top of another block and unstack a block, i.e., move it to the table. We define a predicate on the states of the world, a predicate on the possible actions, a transition function result that maps a state and an action into the resulting state and an interpreter resultlist that takes a state and a plan and returns the state got after executing the plan.

DEFINITION:
nth (n, l)
$=$ **if** $n \simeq 0$ **then** car (l)
else nth $(n - 1, \text{cdr}(l))$ **endif**

DEFINITION:
len (l)
$=$ **if** $l \simeq$ **nil** **then** 0
else $1 +$ len (cdr (l)) **endif**

A block is a litatom.

DEFINITION: blockp (x) = litatom (x)

A stack is a non-empty list of distinct blocks with the first block clear and the last block on the table. We insist that the cdr of the last element be equal to NIL.

DEFINITION:
stackp (l)
$=$ **if** $l \simeq$ **nil** **then** **f**
elseif cdr $(l) =$ **nil** **then** blockp (car (l))
else blockp (car (l)) $\wedge$ (car $(l) \notin$ cdr (l)) $\wedge$ stackp (cdr (l)) **endif**

Two lists are disjoint if they do not have common members.

DEFINITION:
```
disjoint ($l1$, $l2$)
=  if $l1 \simeq$ nil then t
   else (car ($l1$) $\notin$ $l2$) $\land$ disjoint (cdr ($l1$), $l2$) endif
```

A given list is disjoint from each of the members of the given list of lists.

DEFINITION:
```
disjointlist ($s1$, $l1$)
=  if $l1 \simeq$ nil then t
   else disjoint ($s1$, car ($l1$)) $\land$ disjointlist ($s1$, cdr ($l1$)) endif
```

A state is a set of stacks possibly empty. No two stacks have any blocks in common, i.e., they are pairwise disjoint. Once again we insist that the cdr of the last element in the list be NIL.

DEFINITION:
```
bw-statep ($x$)
=  if $x \simeq$ nil
   then if $x$ = nil then t
        else f endif
   else stackp (car ($x$))
        $\land$ disjointlist (car ($x$), cdr ($x$))
        $\land$ bw-statep (cdr ($x$)) endif
```

Here are the predicates and functions on blocks world states.

The predicate ontable. A block is on the table provided it is last on the list of some stack.

Bottomp checks if a block is the bottom block of a given stack.

DEFINITION:
```
bottomp ($b$, $st$)
=  if $st \simeq$ nil then f
   elseif cdr ($st$) $\simeq$ nil then $b$ = car ($st$)
   else bottomp ($b$, cdr ($st$)) endif
```

Find the stack to which a block belongs. This can also be used to assert that a block exists.

DEFINITION:

```
find-stack-of-block ($b$, $s$)
=  if $s \simeq$ nil then f
   elseif $b \in$ car ($s$) then car ($s$)
   else find-stack-of-block ($b$, cdr ($s$)) endif
```

DEFINITION: ontable (b, s) = bottomp (b, find-stack-of-block (b, s))

A block is on top of another block if they appear in sequence in a tower.

The function consecp checks if the given blocks appear in succession in the list of blocks constituting the stack.

DEFINITION:

```
consecp ($b1$, $b2$, $st$)
=  if $st \simeq$ nil then f
   elseif cdr ($st$) $\simeq$ nil then f
   elseif car ($st$) = $b1$ then $b2$ = cadr ($st$)
   else consecp ($b1$, $b2$, cdr ($st$)) endif
```

DEFINITION:
on ($b1$, $b2$, s) = consecp ($b1$, $b2$, find-stack-of-block ($b1$, s))

A block is clear if it is the topmost block of a tower.

DEFINITION: clear (b, s) = assoc (b, s)

Let us go on to actions. We have two actions, move and unstack. The actions are represented as list (`'move`, x, y) and list (`'unstack`, x) where x and y are litatoms that stand for blocks.

We define constructors for the actions as follows:

DEFINITION: move ($b1$, $b2$) = list (`'move`, $b1$, $b2$)

DEFINITION: unstack (b) = list (`'unstack`, b)

We want to define a predicate for plans - sequences of actions. For this we need a predicate for actions.

DEFINITION: movep (x) = (car (x) = `'move`)

DEFINITION: unstackp (x) = (car (x) = `'unstack`)

A plan in the blocks world is a sequence of either move or unstack actions. We allow empty plans.

DEFINITION: actionp (x) = (movep (x) $\vee$ unstackp (x))

DEFINITION:

```
planp (x)
=  if x ≃ nil then t
   else actionp (car (x)) ∧ planp (cdr (x)) endif
```

Specifications of actions. Here are the functions res-move and res-unstack that specify the input-output behavior of the move and unstack actions. These are called by the result function. exec-move is called by res-move if the preconditions of move are satisfied. It deletes the stacks in which b1 and b2 are in s. Then it forms the new stacks and adds them on depending on whether there are blocks under b1 in s.

DEFINITION:

```
delete (x, l)
=  if l ≃ nil then l
   elseif x = car (l) then cdr (l)
   else cons (car (l), delete (x, cdr (l))) endif
```

DEFINITION:

```
exec-move (b1, b2, s)
=  if cdr (assoc (b1, s)) ≃ nil
   then cons (cons (b1, assoc (b2, s)),
              delete (assoc (b1, s), delete (assoc (b2, s), s)))
   else cons (cons (b1, assoc (b2, s)),
             cons (cdr (assoc (b1, s)),
                   delete (assoc (b1, s), delete (assoc (b2, s), s)))) endif
```

Res-move returns an error state when the preconditions of move are not satisfied. Otherwise it calls exec-move. Thus it fully specifies the move action.

DEFINITION:

```
res-move (b1, b2, s)
=  if (b1 = b2) ∨ (¬ clear (b1, s)) ∨ (¬ clear (b2, s))
   then list ('failed, 'res-move, s)
   else exec-move (b1, b2, s) endif
```

Similary exec-unstack returns the state got by unstacking b in s when its preconditions are satisfied.

DEFINITION:
exec-unstack $(b,\ s)$
$=$ **if** cdr (assoc $(b,\ s)$) $\simeq$ **nil** **then** cons (list (b), delete (assoc $(b,\ s)$, s))
else cons (cdr (assoc $(b,\ s)$), cons (list (b), delete (assoc $(b,\ s)$, s))) **endif**

Res-unstack calls exec-unstack when its preconditions are satisfied. Otherwise it returns an error state.

DEFINITION:
res-unstack $(b,\ s)$
$=$ **if** $\neg$ clear $(b,\ s)$ **then** list (`'failed`, `'unstack`, s)
else exec-unstack $(b,\ s)$ **endif**

The function result behaves as follows. If the given state is an error state then it returns it. Otherwise it executes the given action on the given state depending on the action by calling either res-move or res-unstack.

DEFINITION:
result $(a,\ s)$
$=$ **if** car (s) $=$ `'failed` **then** s
elseif movep (a) **then** res-move (cadr (a), caddr (a), s)
else res-unstack (cadr (a), s) **endif**

Resultlist executes a list of actions or plan l in the state s and returns the resulting state.

DEFINITION:
resultlist $(l,\ s)$
$=$ **if** $l \simeq$ **nil** **then** s
else resultlist (cdr (l), result (car (l), s)) **endif**

A.1 Verification of Blocks World Plans

We will first prove that the result of executing legal move and unstack actions in a blocks world state is a legal blocks world state since the proof is quite intricate. Then we will prove that plans achieve goals.

Some rewrite rules for result.

THEOREM: result1
$(\text{bw-statep}\,(s) \land ((\neg\ \text{clear}\,(\mathit{b1},\ s)) \lor (\mathit{b1} = \mathit{b2}) \lor (\neg\ \text{clear}\,(\mathit{b2},\ s))))$
$\rightarrow\ (\text{result}\,(\text{move}\,(\mathit{b1},\ \mathit{b2}),\ s) = \text{list}\,(\texttt{'failed},\ \texttt{'res-move},\ s))$

THEOREM: result5
$(\text{bw-statep}\,(s)$
$\land\ \text{clear}\,(\mathit{b1},\ s)$
$\land\ \text{clear}\,(\mathit{b2},\ s)$
$\land\ (\mathit{b1} \neq \mathit{b2})$
$\land\ (\text{cdr}\,(\text{assoc}\,(\mathit{b1},\ s)) \simeq \textbf{nil}))$
$\rightarrow\ (\text{result}\,(\text{move}\,(\mathit{b1},\ \mathit{b2}),\ s)$
$=\ \text{cons}\,(\text{cons}\,(\mathit{b1},\ \text{assoc}\,(\mathit{b2},\ s)),$
$\text{delete}\,(\text{assoc}\,(\mathit{b1},\ s),\ \text{delete}\,(\text{assoc}\,(\mathit{b2},\ s),\ s))))$

THEOREM: result6
$(\text{bw-statep}\,(s)$
$\land\ \text{clear}\,(\mathit{b1},\ s)$
$\land\ \text{clear}\,(\mathit{b2},\ s)$
$\land\ (\mathit{b1} \neq \mathit{b2})$
$\land\ \text{listp}\,(\text{cdr}\,(\text{assoc}\,(\mathit{b1},\ s))))$
$\rightarrow\ (\text{result}\,(\text{move}\,(\mathit{b1},\ \mathit{b2}),\ s)$
$=\ \text{cons}\,(\text{cons}\,(\mathit{b1},\ \text{assoc}\,(\mathit{b2},\ s)),$
$\text{cons}\,(\text{cdr}\,(\text{assoc}\,(\mathit{b1},\ s)),$
$\text{delete}\,(\text{assoc}\,(\mathit{b1},\ s),\ \text{delete}\,(\text{assoc}\,(\mathit{b2},\ s),\ s)))))$

The value of result for an unstack action.

THEOREM: result2
$(\text{bw-statep}\,(s) \land (\neg\ \text{clear}\,(b,\ s)))$
$\rightarrow\ (\text{result}\,(\text{unstack}\,(b),\ s) = \text{list}\,(\texttt{'failed},\ \texttt{'unstack},\ s))$

THEOREM: result3
$(\text{bw-statep}\,(s) \land \text{clear}\,(b,\ s) \land (\text{cdr}\,(\text{assoc}\,(b,\ s)) \simeq \textbf{nil}))$
$\rightarrow\ (\text{result}\,(\text{unstack}\,(b),\ s) = \text{cons}\,(\text{list}\,(b),\ \text{delete}\,(\text{assoc}\,(b,\ s),\ s)))$

THEOREM: result4
$(\text{bw-statep}\,(s) \land \text{clear}\,(b,\ s) \land \text{listp}\,(\text{cdr}\,(\text{assoc}\,(b,\ s))))$
$\rightarrow\ (\text{result}\,(\text{unstack}\,(b),\ s)$
$=\ \text{cons}\,(\text{cdr}\,(\text{assoc}\,(b,\ s)),\ \text{cons}\,(\text{list}\,(b),\ \text{delete}\,(\text{assoc}\,(b,\ s),\ s))))$

Now we can disable the definitions of result, unstack and move since we have all the info as rewrite rules.

EVENT: Disable result.

EVENT: Disable unstack.

EVENT: Disable move.

We want to prove that if the preconditions are satisfied then the result of doing a move or unstack action is always a blocks world state. The new blocks world state is formed by deleting stacks and adding new stacks to the state. Therefore, we want to prove that deleting two stacks from a set results in a set.

If s1 is disjoint with all the stacks in l1 then it is disjoint with all the stacks got by deleting x from l1.

THEOREM: disj2
$\text{disjointlist}\,(s1,\ l1) \rightarrow \text{disjointlist}\,(s1,\ \text{delete}\,(x,\ l1))$

Deleting a member from a set of stacks leaves it as a set of stacks.

THEOREM: del-set1
$\text{bw-statep}\,(s) \rightarrow \text{bw-statep}\,(\text{delete}\,(x,\ s))$

Deleting the stacks on which b1 and b2 are results in a set of stacks.

THEOREM: set-del-b1-b2
$(\text{bw-statep}\,(s) \wedge \text{clear}\,(b1,\ s) \wedge \text{clear}\,(b2,\ s) \wedge (b1 \neq b2))$
$\rightarrow\ \ \text{bw-statep}\,(\text{delete}\,(\text{assoc}\,(b1,\ s),\ \text{delete}\,(\text{assoc}\,(b2,\ s),\ s)))$

Now we want to prove that adding the new stacks results in a set of stacks.

First we establish that the lists to be added are stacks. Then we prove that they are disjoint from the rest of the stacks in the state got by deleting the original stacks.

THEOREM: assoc-stack1
$(\text{bw-statep}\,(s) \wedge \text{assoc}\,(b,\ s)) \rightarrow \text{stackp}\,(\text{assoc}\,(b,\ s))$

The cdr of a stack is a stack provided it is not empty.

THEOREM: stackp-cdrx1
$(\text{stackp}\,(st) \land \text{listp}\,(\text{cdr}\,(st))) \rightarrow \text{stackp}\,(\text{cdr}\,(st))$

THEOREM: litatom-stack1
$(\text{bw-statep}\,(s) \land (\neg\ \text{litatom}\,(b))) \rightarrow (\neg\ \text{assoc}\,(b,\ s))$

We want to prove that moving a block from st1 to the top of st2 results in a stack provided st1 is disjoint from st2. We do so by proving the following lemma. If there are two distinct stacks st1 and st2 in a state s and there is a block b in st1 then b is not in st2.

Disjointlist means disjoint with every member in the list.

THEOREM: disjoint-mem1
$(\text{disjoint}\,(s1,\ z) \land (x \in s1)) \rightarrow (x \notin z)$

THEOREM: disjointlist-mem1
$(\text{disjointlist}\,(s1,\ l) \land (s2 \in l) \land (x \in s1)) \rightarrow (x \notin s2)$

THEOREM: bex1
$$\begin{array}{ll} & (\text{bw-statep}\,(s) \\ \land & (st1 \in s) \\ \land & (st2 \in s) \\ \land & (st1 \neq st2) \\ \land & (b \in st1)) \\ \rightarrow & (b \notin st2) \end{array}$$

THEOREM: assoc-car1
$\text{assoc}\,(x,\ l) \rightarrow (\text{car}\,(\text{assoc}\,(x,\ l)) = x)$

Establish that b1 and b2 are on distinct stacks.

THEOREM: exec-movel1
$(\text{bw-statep}\,(s) \land \text{clear}\,(b1,\ s) \land \text{clear}\,(b2,\ s) \land (b1 \neq b2))$
$\rightarrow \quad (\text{assoc}\,(b1,\ s) \neq \text{assoc}\,(b2,\ s))$

THEOREM: clear-stackp
$(\text{bw-statep}\,(s) \land \text{clear}\,(b,\ s)) \rightarrow (\text{assoc}\,(b,\ s) \in s)$

Seems to have trouble proving $\text{assoc}\,(b,\ s)$ is a listp.

THEOREM: stack-list1
$(\text{bw-statep}(s) \land \text{clear}(b, s)) \rightarrow \text{listp}(\text{assoc}(b, s))$

THEOREM: exec-movel2
$(\text{bw-statep}(s) \land \text{clear}(b1, s) \land \text{clear}(b2, s) \land (b1 \neq b2))$
$\rightarrow \quad (b1 \notin \text{assoc}(b2, s))$

We can now establish that the new constructs are stacks.

THEOREM: new-stack1
$(\text{bw-statep}(s) \land \text{clear}(b1, s) \land \text{clear}(b2, s) \land (b1 \neq b2))$
$\rightarrow \quad \text{stackp}(\text{cons}(b1, \text{assoc}(b2, s)))$

Now we want to prove that the new stacks are disjoint with the remaining stacks. We prove this by showing that: (a) If a stack is disjointlist with l1 then its cdr is disjointlist with it. (b) If two stacks are disjointlist with l1 then the stack formed by moving the top of one to the top of another is also disjointlist with l1.

If stacks st1 and st2 are disjoint with a stack st3 then the stack formed by moving the top of st1 to the top of st2 is also disjoint with st3.

THEOREM: disj-cons1
$(\text{stackp}(st1)$
$\land \quad \text{stackp}(st2)$
$\land \quad \text{stackp}(st3)$
$\land \quad \text{disjoint}(st1, st3)$
$\land \quad \text{disjoint}(st2, st3))$
$\rightarrow \quad \text{disjoint}(\text{cons}(\text{car}(st1), st2), st3)$

If there are two stacks that are disjoint with a list of stacks s then the stack formed by moving the top of one to the top of another is also disjointlist with s.

THEOREM: disjointlist-cons
$(\text{stackp}(st1)$
$\land \quad \text{stackp}(st2)$
$\land \quad \text{bw-statep}(s)$
$\land \quad \text{disjointlist}(st1, s)$
$\land \quad \text{disjointlist}(st2, s))$
$\rightarrow \quad \text{disjointlist}(\text{cons}(\text{car}(st1), st2), s)$

If a stack st1 is disjoint with another stack st2 then its cdr is also disjoint with it.

THEOREM: disjoint-cdr1
(stackp $(st1)$ ∧ stackp $(st2)$ ∧ disjoint $(st1, st2)$)
→ disjoint (cdr $(st1)$, $st2$)

If a stack st1 is disjointlist with l1 then its cdr is also disjointlist with l1.

THEOREM: disjointlist-cdr
(stackp $(st1)$ ∧ bw-statep $(l1)$ ∧ disjointlist $(st1, l1)$)
→ disjointlist (cdr $(st1)$, $l1$)

Basically we want to prove that a stack is disjoint from the rest of the members of a set of stacks: a key lemma.

The following are lemmas needed to prove that disjoint commutes.

THEOREM: disjoint-nlistp
($s2 \simeq$ **nil**) → disjoint $(s1, s2)$

THEOREM: disjoint-cdr
(car $(s1) \notin s2$) → (disjoint ($s2$, cdr $(s1)$) = disjoint $(s2, s1)$)

Disjoint commutes.

THEOREM: disjoint-comm
disjoint $(s1, s2)$ = disjoint $(s2, s1)$

Lemma: If a stack is deleted from a set of stacks then it is disjointlist with the remaining stacks.

THEOREM: del-disjointlist2
(bw-statep (s) ∧ $(st \in s)$) → disjointlist (st, delete (st, s))

THEOREM: set-del3
(bw-statep (s) ∧ $(st1 \in s)$ ∧ $(st2 \in s)$ ∧ $(st1 \neq st2)$)
→ ($st1 \in$ delete $(st2, s)$)

Now we can prove that the stacks containing b1 and b2 are disjointlist with the set of stacks got by deleting them.

THEOREM: exec-movel5
(bw-statep (s) ∧ clear $(b1, s)$ ∧ clear $(b2, s)$ ∧ $(b1 \neq b2)$)
→ disjointlist (assoc $(b1, s)$, delete (assoc $(b1, s)$, delete (assoc $(b2, s)$, s)))

Theorem: exec-movel6
(bw-statep (s)
 $\land$ clear $(b1, s)$
 $\land$ clear $(b2, s)$
 $\land$ $(b1 \neq b2)$
 $\land$ listp (cdr (assoc $(b1, s))))$
$\rightarrow$ disjointlist (cdr (assoc $(b1, s))$,
 delete (assoc $(b1, s)$, delete (assoc $(b2, s), s)))$

We can prove one case when the rest of the stacks containing b1 is empty. When we add two stacks we must prove that the two are disjoint. These are formed from two existing stacks which we prove are disjoint.

The stacks on which two distinct clear blocks rest are disjoint.

Theorem: set-disjoint1
(bw-statep $(s1) \land (st1 \in s1) \land (st2 \in s1) \land (st1 \neq st2))$
$\rightarrow$ disjoint $(st1, st2)$

Theorem: dist-stacks2
(bw-statep (s) $\land$ clear $(b1, s)$ $\land$ clear $(b2, s) \land (b1 \neq b2))$
$\rightarrow$ disjoint (assoc $(b1, s)$, assoc $(b2, s))$

If two stacks st1 and st2 are disjoint then the stacks got by moving the top of st1 to the top of st2 are also disjoint.

Theorem: disjoint-movel
(stackp $(st1)$ $\land$ stackp $(st2)$ $\land$ disjoint $(st1, st2))$
$\rightarrow$ disjoint (cons (car $(st1)$, $st2)$, cdr $(st1))$

Theorem: disjoint-move2
(stackp $(st1)$ $\land$ stackp $(st2)$ $\land$ disjoint $(st1, st2))$
$\rightarrow$ disjoint (cdr $(st1)$, cons (car $(st1)$, $st2))$

Theorem: exec-movel4
(bw-statep (s) $\land$ clear $(b1, s)$ $\land$ clear $(b2, s) \land (b1 \neq b2))$
$\rightarrow$ disjoint (cdr (assoc $(b1, s))$, cons $(b1$, assoc $(b2, s)))$

The stack formed by b1 on top of the stack assoc $(b2, s)$ is disjoint with the remaining stacks.

THEOREM: exec-movel3
$(\text{bw-statep}(s) \land \text{clear}(b1, s) \land \text{clear}(b2, s) \land (b1 \neq b2))$
$\rightarrow \text{disjointlist}(\text{cons}(b1, \text{assoc}(b2, s)),$
$\quad\quad \text{delete}(\text{assoc}(b1, s), \text{delete}(\text{assoc}(b2, s), s)))$

If there is a state s in which there are distinct blocks b1 and b2 that are clear then moving block b1 to the top of b2 in s results in a blocks world state.

THEOREM: move01
$(\text{bw-statep}(s)$
$\land \text{clear}(b1, s)$
$\land \text{clear}(b2, s)$
$\land (\text{cdr}(\text{assoc}(b1, s)) \simeq \textbf{nil})$
$\land (b1 \neq b2))$
$\rightarrow \text{bw-statep}(\text{result}(\text{move}(b1, b2), s))$

THEOREM: move02
$(\text{bw-statep}(s)$
$\land \text{clear}(b1, s)$
$\land \text{clear}(b2, s)$
$\land \text{listp}(\text{cdr}(\text{assoc}(b1, s)))$
$\land (b1 \neq b2))$
$\rightarrow \text{bw-statep}(\text{result}(\text{move}(b1, b2), s))$

THEOREM: move-sit1
$(\text{bw-statep}(s) \land \text{clear}(b1, s) \land \text{clear}(b2, s) \land (b1 \neq b2))$
$\rightarrow \text{bw-statep}(\text{result}(\text{move}(b1, b2), s))$

We want to prove that unstack also results in a valid blocks world state.

THEOREM: unstackl1
$(\text{bw-statep}(s) \land \text{clear}(b, s) \land \text{listp}(\text{cdr}(\text{assoc}(b, s))))$
$\rightarrow (b \notin \text{cdr}(\text{assoc}(b, s)))$

THEOREM: unstackl2
$(\text{bw-statep}(s) \land \text{clear}(b, s) \land \text{listp}(\text{cdr}(\text{assoc}(b, s))))$
$\rightarrow \text{disjointlist}(\text{cdr}(\text{assoc}(b, s)), \text{delete}(\text{assoc}(b, s), s))$

We want to prove that if a stack is disjointlist with s then the stack formed by its car is also disjointlist with s. We prove the general theorem that if a stack is disjointlist with s then its subsets are also disjointlist with s.

DEFINITION:
subset ($s1$, $s2$)
= **if** $s1 \simeq$ **nil then t**
else (car ($s1$) $\in s2$) $\wedge$ subset (cdr ($s1$), $s2$) **endif**

If a stack st1 is disjoint with a stack st2 then a subset of st1 is also disjoint with st2.

THEOREM: disj-subset
(stackp ($st1$) $\wedge$ stackp ($st2$) $\wedge$ subset ($st3$, $st1$) $\wedge$ disjoint ($st1$, $st2$))
$\rightarrow$ disjoint ($st3$, $st2$)

If there is a stack st that is disjoint with a set of stacks s then every subset of st is also disjoint with s.

THEOREM: disjointlist-subset
(stackp ($st1$) $\wedge$ disjointlist ($st1$, s) $\wedge$ bw-statep (s) $\wedge$ subset ($st2$, $st1$))
$\rightarrow$ disjointlist ($st2$, s)

THEOREM: disjointlist-single
(bw-statep (s) $\wedge$ clear (b, s))
$\rightarrow$ disjointlist (list (b), delete (assoc (b, s), s))

If a clear block is unstacked in a state then it results in a state.

THEOREM: unstack01
(bw-statep (s) $\wedge$ clear (b, s) $\wedge$ (cdr (assoc (b, s)) $\simeq$ **nil**))
$\rightarrow$ bw-statep (result (unstack (b), s))

THEOREM: unstack02
(bw-statep (s) $\wedge$ clear (b, s) $\wedge$ listp (cdr (assoc (b, s))))
$\rightarrow$ bw-statep (result (unstack (b), s))

THEOREM: unstack-sit1
(bw-statep (s) $\wedge$ clear (b, s)) $\rightarrow$ bw-statep (result (unstack (b), s))

Having proved that the move and the unstack actions result in legal blocks world states, we turn our attention to verifying plans. Plans are expressed by plan generating Lisp programs and proved correct.

Example 1 : Plan to clear a block. The following lemmas are needed to establish admissibility of makeclear-gen. The measure len (find-stack-of-block (b, s)) decreases with every recursive call.

THEOREM: mem-find0
$(\text{bw-statep}\,(s) \wedge \text{find-stack-of-block}\,(b,\ s))$
$\rightarrow\ \ (\text{find-stack-of-block}\,(b,\ s) \in s)$

THEOREM: mem-find6
$(\text{bw-statep}\,(s) \wedge \text{find-stack-of-block}\,(b,\ s))$
$\rightarrow\ \ (b \in \text{find-stack-of-block}\,(b,\ s))$

THEOREM: find-mem-cdr
$(\text{bw-statep}\,(s)$
$\wedge\ \ \text{find-stack-of-block}\,(b,\ s)$
$\wedge\ \ (b \neq \text{car}\,(\text{find-stack-of-block}\,(b,\ s))))$
$\rightarrow\ \ (b \in \text{cdr}\,(\text{find-stack-of-block}\,(b,\ s)))$

Here is a theorem about the effect of unstack that we need. If I unstack the top of a non-empty stack in a state then the cdr of the stack is a member of the resulting state.

THEOREM: clear-car1
$(\text{bw-statep}\,(s) \wedge (st \in s)) \rightarrow \text{assoc}\,(\text{car}\,(st),\ s)$

THEOREM: disjoint-car1
$(\text{stackp}\,(st1) \wedge \text{stackp}\,(st2) \wedge \text{disjoint}\,(st1,\ st2))$
$\rightarrow\ \ (\text{car}\,(st1) \neq \text{car}\,(st2))$

THEOREM: state-stackp
$(\text{bw-statep}\,(s) \wedge (\neg\ \text{stackp}\,(st))) \rightarrow (st \notin s)$

THEOREM: not-eq-car1
$(\text{bw-statep}\,(s) \wedge (st1 \in s) \wedge (st2 \in s) \wedge (st1 \neq st2))$
$\rightarrow\ \ (\text{car}\,(st1) \neq \text{car}\,(st2))$

THEOREM: not-eq-car2
$(\text{bw-statep}\,(s) \wedge (st \in s) \wedge (st \neq \text{car}\,(s))) \rightarrow (\text{car}\,(st) \neq \text{caar}\,(s))$

THEOREM: state-cdr1
$(\text{listp}\,(s) \wedge \text{bw-statep}\,(s)) \rightarrow \text{bw-statep}\,(\text{cdr}\,(s))$

This is an important lemma. This states that if there is a stack st in s then trying to find the stack whose car is the top of st will result in st.

THEOREM: assoc-car2
$(\text{bw-statep}(s) \land (st \in s)) \rightarrow (\text{assoc}(\text{car}(st), s) = st)$

THEOREM: result7
$(\text{bw-statep}(s) \land (st \in s) \land \text{listp}(\text{cdr}(st)))$
$\rightarrow\ (\text{result}(\text{unstack}(\text{car}(st)), s)$
$\quad = \text{cons}(\text{cdr}(st), \text{cons}(\text{list}(\text{car}(st)), \text{delete}(st, s))))$

THEOREM: unstack-eff1
$(\text{bw-statep}(s) \land (st \in s) \land \text{listp}(\text{cdr}(st)))$
$\rightarrow\ (\text{cdr}(st) \in \text{result}(\text{unstack}(\text{car}(st)), s))$

EVENT: Disable bw-statep.

Proving find-assoc1

THEOREM: disjointlist-mem2
$(\text{disjointlist}(s1, l) \land (s2 \in l) \land (x \in s2)) \rightarrow (x \notin s1)$

THEOREM: find-stack3
$(\text{bw-statep}(s) \land (st \in s) \land (b \in st))$
$\rightarrow\ (\text{find-stack-of-block}(b, s) = st)$

If a block is clear then find-stack returns the same stack as does assoc. This is because we know b belongs to assoc (b, s) and by mem-find3 it follows.

THEOREM: clear-mem1
$(\text{bw-statep}(s) \land \text{clear}(b, s)) \rightarrow (b \in \text{assoc}(b, s))$

THEOREM: find-assoc1
$(\text{bw-statep}(s) \land \text{clear}(b, s)) \rightarrow (\text{find-stack-of-block}(b, s) = \text{assoc}(b, s))$

THEOREM: find5
$(\text{bw-statep}(s)$
$\land\ \text{find-stack-of-block}(b, s)$
$\land\ (b \neq \text{car}(\text{find-stack-of-block}(b, s))))$
$\rightarrow\ (\text{find-stack-of-block}(b,$
$\qquad\qquad \text{result}(\text{unstack}(\text{car}(\text{find-stack-of-block}(b, s))), s))$
$\quad = \text{cdr}(\text{find-stack-of-block}(b, s)))$

THEOREM: find-stack-listp
(bw-statep (s) $\wedge$ find-stack-of-block $(b,\ s)$)
$\rightarrow$ listp (find-stack-of-block $(b,\ s)$)

DEFINITION: m3 $(b,\ s)$ = len (find-stack-of-block $(b,\ s)$)

Plan to clear a block by unstacking one by one the blocks on top of it.

DEFINITION:
```
makeclear-gen (b, s)
=  if (¬ find-stack-of-block (b, s)) ∨ (¬ bw-statep (s))  then f
   elseif b = car (find-stack-of-block (b, s))  then nil
   else cons (unstack (car (find-stack-of-block (b, s))),
              makeclear-gen (b,
                             result (unstack (car (find-stack-of-block (b,
                                                                       s))),
                                     s))) endif
```

Proving that plans generated by makeclear-gen achieve the desired goal.

THEOREM: mkc-l1
(bw-statep (s)
$\wedge$ find-stack-of-block $(b,\ s)$
$\wedge$ (car (find-stack-of-block $(b,\ s)$) $= b$))
$\rightarrow$ assoc $(b,\ s)$

THEOREM: mkc-l2
(bw-statep (s)
$\wedge$ find-stack-of-block $(b,\ s)$
$\wedge$ ($b \neq$ car (find-stack-of-block $(b,\ s)$)))
$\rightarrow$ find-stack-of-block (b, result (unstack (car (find-stack-of-block $(b,\ s)$)), s))

THEOREM: makeclear-works
(bw-statep (s) $\wedge$ find-stack-of-block $(b,\ s)$)
$\rightarrow$ clear (b, resultlist (makeclear-gen $(b,\ s)$, s))

Makeclear-gen generates valid plans.

THEOREM: make-clear-is-a-plan
planp (makeclear-gen $(b,\ s)$)

Makeclear-gen results in a legal blocks world state.

THEOREM: makeclear-bwstate
(bw-statep (s) $\wedge$ find-stack-of-block $(b,\ s)$)
$\rightarrow$ bw-statep (resultlist (makeclear-gen $(b,\ s)$, s))

THEOREM: makeclear-works1
(bw-statep (s)
$\wedge$ find-stack-of-block $(b,\ s)$
$\wedge$ ($p1$ = makeclear-gen $(b,\ s)$)
$\wedge$ ($s1$ = resultlist $(p1,\ s)$))
$\rightarrow$ (bw-statep $(s1)$ $\wedge$ planp $(p1)$ $\wedge$ clear $(b,\ s1)$)

We would also like to prove that makeclear-gen preserves the set of blocks that exist in its initial state. The proof of this was not carried out. But the following definitions of set-equal and set-of-blocks were used in the statement of the theorem given in the dissertation.

```
DEFINITION:
set-equal ($s1$, $s2$)
=  **if** $s1 \simeq$ **nil then** $s2 \simeq$ **nil**
   **else** (car ($s1$) $\in$ $s2$) $\wedge$ set-equal (cdr ($s1$), delete (car ($s1$), $s2$)) **endif**
```

```
DEFINITION:
set-of-blocks ($s$)
=  **if** $s \simeq$ **nil then nil**
   **else** append (car ($s$), set-of-blocks (cdr ($s$))) **endif**
```

Example 2 - Plan to invert one tower on top of another. Invert-gen is a plan generator to invert one tower over the top of another.

```
DEFINITION:
invert-gen ($st1$, $st2$, $s$)
=  **if** $st1 \simeq$ **nil then nil**
   **else** cons (move (car ($st1$), car ($st2$)),
                 invert-gen (cdr ($st1$),
                             cons (car ($st1$), $st2$),
                             result (move (car ($st1$), car ($st2$)), $s$))) **endif**
```

```
DEFINITION:
reverse ($l$)
=  **if** $l \simeq$ **nil then nil**
   **else** append (reverse (cdr ($l$)), list (car ($l$))) **endif**
```

Theorem: mem-cadr1
$x \in \text{cons}\,(y,\ \text{cons}\,(x,\ z))$

Theorem: mem-car1
$x \in \text{cons}\,(x,\ y)$

Theorem: result9
$$\begin{array}{l}
(\text{bw-statep}\,(s) \\
\ \wedge\ \ (\mathit{st1} \in s) \\
\ \wedge\ \ (\mathit{st2} \in s) \\
\ \wedge\ \ \text{listp}\,(\text{cdr}\,(\mathit{st1})) \\
\ \wedge\ \ (\mathit{st1} \neq \mathit{st2})) \\
\to\ \ (\text{result}\,(\text{move}\,(\text{car}\,(\mathit{st1}),\ \text{car}\,(\mathit{st2})),\ s) \\
\qquad = \ \ \text{cons}\,(\text{cons}\,(\text{car}\,(\mathit{st1}),\ \mathit{st2}), \\
\qquad\qquad\quad \text{cons}\,(\text{cdr}\,(\mathit{st1}),\ \text{delete}\,(\mathit{st1},\ \text{delete}\,(\mathit{st2},\ s)))))
\end{array}$$

Theorem: result10
$$\begin{array}{l}
(\text{bw-statep}\,(s) \\
\ \wedge\ \ (\mathit{st1} \in s) \\
\ \wedge\ \ (\mathit{st2} \in s) \\
\ \wedge\ \ (\text{cdr}\,(\mathit{st1}) \simeq \mathbf{nil}) \\
\ \wedge\ \ (\mathit{st1} \neq \mathit{st2})) \\
\to\ \ (\text{result}\,(\text{move}\,(\text{car}\,(\mathit{st1}),\ \text{car}\,(\mathit{st2})),\ s) \\
\qquad = \ \ \text{cons}\,(\text{cons}\,(\text{car}\,(\mathit{st1}),\ \mathit{st2}),\ \text{delete}\,(\mathit{st1},\ \text{delete}\,(\mathit{st2},\ s))))
\end{array}$$

Theorem: move-eff1
$$\begin{array}{l}
(\text{bw-statep}\,(s) \\
\ \wedge\ \ (\mathit{st1} \in s) \\
\ \wedge\ \ (\mathit{st2} \in s) \\
\ \wedge\ \ \text{listp}\,(\text{cdr}\,(\mathit{st1})) \\
\ \wedge\ \ (\mathit{st1} \neq \mathit{st2})) \\
\to\ \ (\text{cdr}\,(\mathit{st1}) \in \text{result}\,(\text{move}\,(\text{car}\,(\mathit{st1}),\ \text{car}\,(\mathit{st2})),\ s))
\end{array}$$

Theorem: move-eff21
$$\begin{array}{l}
(\text{bw-statep}\,(s) \\
\ \wedge\ \ (\mathit{st1} \in s) \\
\ \wedge\ \ (\mathit{st2} \in s) \\
\ \wedge\ \ (\text{cdr}\,(\mathit{st1}) \simeq \mathbf{nil}) \\
\ \wedge\ \ (\mathit{st1} \neq \mathit{st2})) \\
\to\ \ (\text{cons}\,(\text{car}\,(\mathit{st1}),\ \mathit{st2}) \in \text{result}\,(\text{move}\,(\text{car}\,(\mathit{st1}),\ \text{car}\,(\mathit{st2})),\ s))
\end{array}$$

THEOREM: move-eff22
(bw-statep (s)
$\land$ $(st1 \in s)$
$\land$ $(st2 \in s)$
$\land$ listp (cdr $(st1)$)
$\land$ $(st1 \neq st2)$)
$\to$ (cons (car $(st1)$, $st2$) $\in$ result (move (car $(st1)$, car $(st2)$), s))

THEOREM: move-eff2
(bw-statep (s) $\land$ $(st1 \in s)$ $\land$ $(st2 \in s)$ $\land$ $(st1 \neq st2)$)
$\to$ (cons (car $(st1)$, $st2$) $\in$ result (move (car $(st1)$, car $(st2)$), s))

EVENT: Disable disjoint-comm.

EVENT: Disable disjoint-cdr.

THEOREM: not-eq-l3
(listp $(st1)$ $\land$ listp $(st2)$ $\land$ disjoint $(st1, st2)$)
$\to$ (cdr $(st1)$ $\neq$ cons (car $(st1)$, $st2$))

THEOREM: not-eq-st1-st2
(bw-statep (s) $\land$ $(st1 \in s)$ $\land$ $(st2 \in s)$ $\land$ $(st1 \neq st2)$)
$\to$ (cdr $(st1)$ $\neq$ cons (car $(st1)$, $st2$))

THEOREM: bw-state1
(bw-statep (s) $\land$ $(st1 \in s)$ $\land$ $(st2 \in s)$ $\land$ $(st1 \neq st2)$)
$\to$ bw-statep (result (move (car $(st1)$, car $(st2)$), s))

THEOREM: append-assoc1
append (append (x, y), z) = append (x, append (y, z))

THEOREM: invert-works1
(bw-statep (s)
$\land$ $(st1 \in s)$
$\land$ $(st2 \in s)$
$\land$ (cdr $(st1)$ $\simeq$ **nil**)
$\land$ $(st1 \neq st2)$)
$\to$ (cons (car $(st1)$, $st2$) $\in$ resultlist (invert-gen $(st1, st2, s)$, s))

Definition:

```
f6 $(st1, st2, s)$
= if $st1 \simeq$ nil then t
  elseif cdr $(st1) \simeq$ nil then t
  else f6 (cdr $(st1)$,
           cons (car $(st1)$, $st2$),
           result (move (car $(st1)$, car $(st2)$), $s$)) endif
```

Theorem: invert-works

$(\text{bw-statep}(s) \land (st1 \in s) \land (st2 \in s) \land (st1 \neq st2))$
$\rightarrow (\text{append}(\text{reverse}(st1), st2) \in \text{resultlist}(\text{invert-gen}(st1, st2, s), s))$

Invert-gen generates valid plans.

Theorem: invert-isa-plan

$\text{planp}(\text{invert-gen}(st1, st2, s))$

Invert-gen plans results in a valid blocks world state.

Theorem: bw-state2

$(\text{bw-statep}(s)$
$\land\ (\text{cons}(z, x) \in s)$
$\land\ (\text{cons}(v, w) \in s)$
$\land\ (\text{cons}(z, x) \neq \text{cons}(v, w)))$
$\rightarrow\ \text{bw-statep}(\text{result}(\text{move}(z, v), s))$

Theorem: invert-resultsin-state

$(\text{bw-statep}(s) \land (st1 \in s) \land (st2 \in s) \land (st1 \neq st2))$
$\rightarrow\ \text{bw-statep}(\text{resultlist}(\text{invert-gen}(st1, st2, s), s))$

Invert-gen works.

Theorem: invert-works2

$(\text{bw-statep}(s)$
$\land\ (st1 \in s)$
$\land\ (st2 \in s)$
$\land\ (st1 \neq st2)$
$\land\ (p1 = \text{invert-gen}(st1, st2, s))$
$\land\ (s1 = \text{resultlist}(p1, s)))$
$\rightarrow\ (\text{bw-statep}(s1) \land \text{planp}(p1) \land (\text{append}(\text{reverse}(st1), st2) \in s1))$

Example 3. Plan to unstack all the blocks in a given tower.

DEFINITION:
```
unstack-tower-gen $(st, s)$
$=$ if $st \simeq$ nil then nil
    elseif cdr $(st) \simeq$ nil then nil
    else cons (unstack (car $(st)$),
                unstack-tower-gen (cdr $(st)$, result (unstack (car $(st)$), $s$))) endif
```

The following checks if all blocks of stack st are on the table and clear in the state s.

DEFINITION:
```
ontable-list $(st, s)$
$=$ if $st \simeq$ nil then t
    else (find-stack-of-block (car $(st)$, $s$) $=$ list (car $(st)$))
         $\land$ ontable-list (cdr $(st)$, $s$) endif
```

The following events prove unstack-tower-gen works.

THEOREM: result8
(bw-statep (s) $\land$ $(st \in s)$ $\land$ (cdr $(st) \simeq$ **nil**))
$\rightarrow$ (result (unstack (car (st)), s) $=$ cons (list (car (st)), delete (st, s)))

Unstack stack st2 leaves stack st1 undisturbed.

THEOREM: unstack-eff41
(bw-statep (s)
$\land$ $(st1 \in s)$
$\land$ $(st2 \in s)$
$\land$ (cdr $(st2) \simeq$ **nil**)
$\land$ $(st1 \neq st2)$)
$\rightarrow$ $(st1 \in$ result (unstack (car $(st2)$), s))

THEOREM: unstack-eff42
(bw-statep (s)
$\land$ $(st1 \in s)$
$\land$ $(st2 \in s)$
$\land$ listp (cdr $(st2)$)
$\land$ $(st1 \neq st2)$)
$\rightarrow$ $(st1 \in$ result (unstack (car $(st2)$), s))

THEOREM: unstack-eff4
(bw-statep (s) $\land$ $(st1 \in s)$ $\land$ $(st2 \in s)$ $\land$ $(st1 \neq st2)$)
$\rightarrow$ $(st1 \in$ result (unstack (car $(st2)$), s))

THEOREM: unstack-eff6
$(\text{bw-statep}(s) \land (st \in s))$
$\rightarrow \quad (\text{list}(\text{car}(st)) \in \text{result}(\text{unstack}(\text{car}(st)), s))$

DEFINITION:
f3 (st, s)

```
= if $st \simeq$ nil then t
  elseif cdr $(st) \simeq$ nil then t
  else f3 (cdr $(st)$, result (unstack (car $(st)$), $s$)) endif
```

THEOREM: unstack-tower-eff5
$(\text{bw-statep}(s) \land (st1 \in s) \land (b \notin st1) \land (\text{list}(b) \in s))$
$\rightarrow \quad (\text{find-stack-of-block}(b, \text{resultlist}(\text{unstack-tower-gen}(st1, s), s)) = \text{list}(b))$

THEOREM: mem-cdr3
$(\text{bw-statep}(s) \land (st \in s)) \rightarrow (\text{car}(st) \notin \text{cdr}(st))$

THEOREM: unstack-tower-works
$(\text{bw-statep}(s) \land (st \in s))$
$\rightarrow \quad \text{ontable-list}(st, \text{resultlist}(\text{unstack-tower-gen}(st, s), s))$

THEOREM: unstack-tower-isa-plan
$\text{planp}(\text{unstack-tower-gen}(st, s))$

THEOREM: unstack-tower-eff51
$(\text{bw-statep}(s) \land (st \in s) \land \text{listp}(\text{cdr}(st)))$
$\rightarrow \quad (\text{find-stack-of-block}(\text{car}(st), \text{resultlist}(\text{unstack-tower-gen}(\text{cdr}(st), \text{result}(\text{unstack}(\text{car}(st)), s)), \text{result}(\text{unstack}(\text{car}(st)), s))) = \text{list}(\text{car}(st)))$

THEOREM: unstack-tower-state
$(\text{bw-statep}(s) \land (st \in s))$
$\rightarrow \quad \text{bw-statep}(\text{resultlist}(\text{unstack-tower-gen}(st, s), s))$

Unstack-tower-gen works.

THEOREM: unstack-tower-works1

```
 (bw-statep (s)
  ∧  (st ∈ s)
  ∧  (p1 = unstack-tower-gen (st, s))
  ∧  (s1 = resultlist (p1, s)))
 →  (bw-statep (s1) ∧ planp (p1) ∧ ontable-list (st, s1))
```

EVENT: Enable bw-statep.

EVENT: Enable disjoint-comm.

EVENT: Enable disjoint-cdr.

Example 4. Plan to unstack all towers so that all the blocks are on the table and clear.

The function get-block goes through a blocks world state and returns a block that is not on the table. Otherwise it returns F. Thus, if get-block returns F all the blocks in the state are on the table and (hence) clear.

DEFINITION:

```
get-block (s)
=  if s ≃ nil then f
   elseif ¬ bw-statep (s) then f
   elseif cdar (s) ≃ nil then get-block (cdr (s))
   else caar (s) endif
```

Unstack-all-towers-gen is a plan to unstack all the blocks in a state so that they are on the table. We want to prove that it will terminate, i.e., we must prove that get-block (s) will eventually become false. We use the difference in the number of blocks and the number of singletons in a state as a metric.

DEFINITION:

```
number-of-blocks (s)
=  if s ≃ nil then 0
   else len (car (s)) + number-of-blocks (cdr (s)) endif
```

DEFINITION:

```
number-of-singletons (s)
=  if s ≃ nil then 0
   elseif cdar (s) = nil then 1 + number-of-singletons (cdr (s))
   else number-of-singletons (cdr (s)) endif
```

Definition:
m1 (s) = (number-of-blocks (s) − number-of-singletons (s))

To establish admissibility we must prove that the number of blocks in a state remains unchanged after an unstack operation.

Theorem: gb1
get-block $(s) \rightarrow$ assoc (get-block (s), s)

Theorem: gb2
get-block $(s) \rightarrow$ listp (cdr (assoc (get-block (s), s)))

Theorem: result-get-block
get-block (s)
$\rightarrow$ (result (unstack (get-block (s)), s)
    = cons (cdr (assoc (get-block (s), s)),
        cons (list (get-block (s)), delete (assoc (get-block (s), s), s))))

Theorem: num1
(($st \in s$) $\wedge$ listp (cdr (st)))
$\rightarrow$ ((len (cdr (st)) + `1` + number-of-blocks (delete (st, s)))
    = number-of-blocks (s))

Theorem: mem-assoc1
assoc (x, y) $\rightarrow$ (assoc (x, y) $\in y$)

Theorem: unstack-leaves-numblocks-unchanged
get-block (s)
$\rightarrow$ (number-of-blocks (result (unstack (get-block (s)), s))
    = number-of-blocks (s))

Theorem: num2
(($st \in s$) $\wedge$ listp (cdr (st)))
$\rightarrow$ (number-of-singletons (delete (st, s)) = number-of-singletons (s))

Theorem: unstack-increases-num-singletons
get-block (s)
$\rightarrow$ (number-of-singletons (s)
    $<$ number-of-singletons (result (unstack (get-block (s)), s)))

THEOREM: num31
$((y1 < y2) \wedge (y2 \leq x2) \wedge (x1 = x2))$
$\rightarrow\ (((x2 - y2) < (x1 - y1)) = \mathbf{t})$

THEOREM: nsing-leq-nblocks
(number-of-singletons (s) $\leq$ number-of-blocks $(s)) = \mathbf{t}$

THEOREM: nsing-less-than-nblocks
get-block $(s) \rightarrow$ (number-of-singletons (s) $<$ number-of-blocks (s))

THEOREM: adm1
get-block $(s) \rightarrow$ (m1 (result (unstack (get-block (s)), s)) $<$ m1 (s))

EVENT: Disable m1.

EVENT: Disable result-get-block.

DEFINITION:
```
unstack-all-towers-gen (s)
=  if get-block (s)
   then cons (unstack (get-block (s)),
              unstack-all-towers-gen (result (unstack (get-block (s)), s)))
   else nil endif
```

All the towers in a state can be unstacked.

THEOREM: unstackall1
bw-statep $(s) \rightarrow (\neg$ get-block (resultlist (unstack-all-towers-gen (s), s)))

The resulting state is a blocks world state.

THEOREM: get-block-litatom
(bw-statep $(s) \wedge$ get-block $(s)) \rightarrow$ litatom (get-block (s))

THEOREM: unstackall2
bw-statep $(s) \rightarrow$ bw-statep (resultlist (unstack-all-towers-gen (s), s))

unstack-all-towers-gen generates plans.

THEOREM: unstackall3
planp (unstack-all-towers-gen (s))

Unstack-all-towers-gen works.

THEOREM: unstack-all-towers-works
(bw-statep (s)
$\wedge$ ($p1$ = unstack-all-towers-gen (s))
$\wedge$ ($s1$ = resultlist ($p1$, s)))
$\rightarrow$ (bw-statep ($s1$) $\wedge$ planp ($p1$) $\wedge$ ($\neg$ get-block ($s1$)))

Here are examples from Chapter 2 of theorems on concrete data. I have also included statements of theorems that were not verified.

Example of a blocks world state.

DEFINITION: STATE1 = `'((a b c) (d e))`

THEOREM: blocks-world-example1
bw-statep (STATE1)

Find-stack-of-block examples.

THEOREM: find-stack-example1
find-stack-of-block (`'c`, STATE1) = `'(a b c)`

THEOREM: find-stack-example2
$\neg$ find-stack-of-block (`'g`, STATE1)

Plan example.

DEFINITION:
PLAN1 = `'((unstack a) (move b a) (move c b))`

THEOREM: planp-example1
planp (PLAN1)

Examples of moving and unstacking blocks both legal and illegal.

THEOREM: legal-move-example
result (move (`'a`, `'d`), STATE1) = `'((a d e) (b c))`

THEOREM: legal-unstack-example
result (unstack (`'a`), STATE1) = `'((b c) (a) (d e))`

THEOREM: illegal-move-example
result (move (`'b`, `'d`), STATE1)
= `'(failed res-move ((a b c) (d e)))`

THEOREM: illegal-unstack-example
result (unstack (`'e`), STATE1) = `'(failed unstack ((a b c) (d e)))`

Resultlist examples

THEOREM: resultlist-legal
resultlist (PLAN1, STATE1) = `'((c b a) (d e))`

THEOREM: resultlist-illegal
resultlist (`'((unstack b) (move a c))`, STATE1)
= `'(failed unstack ((a b c) (d e)))`

Makeclear example on a concrete state. There exists a sequence of actions to clear block 'C in state1.

THEOREM: makeclear-gen-ex1
$((p =$ makeclear-gen (`'c`, STATE1)$) \land (s =$ resultlist $(p,$ STATE1$)))$
$\rightarrow$ (bw-statep $(s) \land$ planp $(p) \land$ clear (`'c`, s))

THEOREM: invert-gen-ex1
$((p1 =$ invert-gen (`'(a b c)`, `'(d e)`, STATE1$)$)
$\land$ $(s1 =$ resultlist $(p1,$ STATE1$)))$
$\rightarrow$ (bw-statep $(s1) \land$ planp $(p1) \land$ (`'(c b a d e)` $\in s1$))

An example with arithmetic. Given a state in which there are at least n stacks, show that we can achieve a state in which there is a stack of height n.

The plan to form the stack is to use the topmost blocks of all the towers.

DEFINITION:
```
build-towern (s, n)
= if n ≃ 0 then nil
  elseif n = 1 then list (unstack (caar (s)))
  else append (build-towern (cdr (s), n − 1),
               list (move (caar (s), caadr (s)))) endif
```

DEFINITION:
exists-towern (n, s)
$=$ **if** $s \simeq$ **nil then f**
  elseif len (car (s)) $= n$ **then t**
  else exists-towern $(n$, cdr $(s))$ **endif**

Testing build-towern on a concrete example.

THEOREM: build-towern-ex1
build-towern (STATE1, 2) = `'((unstack d) (move a d))`

THEOREM: build-towern-ex2
exists-towern (2, resultlist (build-towern (STATE1, 2), STATE1))

The strategy to go from any blocks world state to any other.

Form-tower returns the sequence of actions that will form the stack from a state in which all the blocks in the stack are on the table.

DEFINITION:
form-tower (st)
$=$ **if** $(st \simeq$ **nil**$) \vee ($cdr $(st) \simeq$ **nil**$)$ **then nil**
  else append (form-tower (cdr (st)), list (move (car (st), cadr (st)))) **endif**

Form state s2 from s1. Form-state works by forming all the towers in the state. The initial state s1 has all the blocks on the table.

DEFINITION:
form-state $(s2)$
$=$ **if** $s2 \simeq$ **nil then nil**
  else append (form-tower (car $(s2)$), form-state (cdr $(s2)$)) **endif**

DEFINITION:
transform $(s1, s2)$ = append (unstack-all-towers-gen $(s1)$, form-state $(s2)$)

The following theorems were not checked by the theorem prover. †

THEOREM: makclear-gen-preserves-blocks
(bw-statep $(s0)$
 $\wedge$ find-stack-of-block $(b, s0)$
 $\wedge$ $(p1 =$ makeclear-gen $(b, s0))$
 $\wedge$ $(s1 =$ resultlist $(p1, s0)))$
$\rightarrow$ set-equal (set-of-blocks $(s0)$, set-of-blocks $(s1)$)

THEOREM: build-towern-works
(bw-statep (s)
 $\land$ (len $(s) > n)$
 $\land$ $(n \not\simeq 0)$
 $\land$ $(p1 =$ build-towern $(s, n))$
 $\land$ $(s1 =$ resultlist $(p1, s)))$
 $\rightarrow$ (bw-statep $(s1) \land$ planp $(p1) \land$ exists-towern $(n, s1))$

THEOREM: planner-works
(bw-statep $(s1)$
 $\land$ bw-statep $(s2)$
 $\land$ set-equal (set-of-blocks $(s1)$, set-of-blocks $(s2)))$
 $\rightarrow$ (resultlist (transform $(s1, s2), s1) = s2)$

A.2 Blocks World with Side-Effects

The following sequence of events formalizes the blocks world in which there is a single move action that has the side-effect of moving all the blocks on top of the block being moved along with it. We use the blocks world formalization to verify mechanically a plan generating program that forms a single tower out of all the blocks in the initial state. We loaded this sequence of events after we loaded the events in appendix A upto the event called find-stack-listp in Section A.1. These contain the necessary properties of states needed for this proof. To obtain those events, I did a (ubt m3) after loading the above events before loading this file.

The following function returns the list of blocks in a stack above a given block starting at the top and ending with the block.

DEFINITION:
```
stack-above ($b$, $st$)
=  **if** $st \simeq$ **nil then nil**
   **elseif** $b =$ car ($st$) **then** list (car ($st$))
   **else** cons (car ($st$), stack-above ($b$, cdr ($st$))) **endif**
```

DEFINITION:
stack-above1 $(x, s) =$ stack-above $(x,$ find-stack-of-block $(x, s))$

We can say x is over y in s if x belongs to the stack above y in s.

DEFINITION:
over $(x, y, s) = ((x \neq y) \land (x \in$ stack-above1 $(y, s)))$

Need another function stack-below.

DEFINITION:

```
stack-below $(x,\ st)$
= if $st \simeq$ nil then f
  elseif $x =$ car $(st)$ then cdr $(st)$
  else stack-below $(x,$ cdr $(st))$ endif
```

DEFINITION:

stack-below1 $(x,\ s)$ = stack-below $(x,$ find-stack-of-block $(x,\ s))$

Exec-move-over implements the effects of the move over action.

DEFINITION:

```
exec-move-over $(x,\ y,\ s)$
= if stack-below1 $(x,\ s) \simeq$ nil
  then cons (append (stack-above1 $(x,\ s)$, assoc $(y,\ s)$),
              delete (find-stack-of-block $(x,\ s)$, delete (assoc $(y,\ s)$, $s$)))
  else cons (stack-below1 $(x,\ s)$,
             cons (append (stack-above1 $(x,\ s)$, assoc $(y,\ s)$),
                   delete (find-stack-of-block $(x,\ s)$,
                           delete (assoc $(y,\ s)$, $s$)))) endif
```

To move x over y, y must be clear and y and x should be on distinct stacks. x and y must exist.

DEFINITION:

```
res-move-over $(x,\ y,\ s)$
= if $(\neg$ find-stack-of-block $(x,\ s))$
      $\lor$  (find-stack-of-block $(x,\ s) =$ assoc $(y,\ s)$)
      $\lor$  $(\neg$ clear $(y,\ s))$ then list ('failed, 'move-over, $x$, $y$, $s$)
  else exec-move-over $(x,\ y,\ s)$ endif
```

Constructor for the action move-over.

DEFINITION: move-over $(x,\ y)$ = list ('move-over, x, y)

DEFINITION: move-overp (x) = (car (x) = 'move-over)

A new result function.

DEFINITION:

```
result-new $(a,\ s)$
= if car $(s)$ = 'failed then $s$
  else res-move-over (cadr $(a)$, caddr $(a)$, $s$) endif
```

The state got from s as a result of doing a list of actions.

DEFINITION:
resultlist-new $(l,\ s)$
$=$ **if** $l \simeq$ **nil then** s
else resultlist-new (cdr (l), result-new (car (l), s)) **endif**

We want to show that there is a plan to form a single tower. Move the last block of the first tower to the top of the second tower and so on until there is exactly one tower.

DEFINITION:
last (l)
$=$ **if** $l \simeq$ **nil then f**
elseif cdr $(l) \simeq$ **nil then** car (l)
else last (cdr (l)) **endif**

EVENT: Disable disjoint-comm.

EVENT: Disable disjoint-cdr.

EVENT: Disable bw-statep.

To get the following function admitted we need the following lemmas.

THEOREM: mem-last1
stackp $(st) \rightarrow$ (last $(st) \in st$)

THEOREM: single-l2
stackp $(st) \rightarrow$ (stack-below (last (st), st) $=$ **nil**)

THEOREM: single-l3
stackp $(st) \rightarrow$ (stack-above (last (st), st) $= st$)

THEOREM: result-new1
(bw-statep (s)
$\wedge$ find-stack-of-block $(x,\ s)$
$\wedge$ (find-stack-of-block $(x,\ s) \neq$ assoc $(y,\ s)$)
$\wedge$ clear $(y,\ s)$
$\wedge$ (stack-below1 $(x,\ s) \simeq$ **nil**))
$\rightarrow$ (result-new (move-over $(x,\ y)$, s)
$=$ cons (append (stack-above1 $(x,\ s)$, assoc $(y,\ s)$),
delete (find-stack-of-block $(x,\ s)$, delete (assoc $(y,\ s)$, s))))

EVENT: Disable result-new.

EVENT: Disable move-over.

The following are dumb lemmas that I am proving so that the theorem prover can find them easily.

THEOREM: l5
(bw-statep(s) $\wedge$ listp (cdr(s)))
$\rightarrow$ (find-stack-of-block (last (car(s)), s) = car(s))

THEOREM: l6
(bw-statep(s) $\wedge$ listp (cdr(s))) $\rightarrow$ (car(s) $\neq$ cadr(s))

THEOREM: l7
(bw-statep(s) $\wedge$ listp (cdr(s)))
$\rightarrow$ (stack-below (last (car(s)), car(s)) = **nil**)

THEOREM: l8
(bw-statep(s) $\wedge$ listp (cdr(s)))
$\rightarrow$ (stack-above (last (car(s)), car(s)) = car(s))

THEOREM: l9
listp (cdr(s)) $\rightarrow$ (delete (car(s), delete (cadr(s), s)) = cddr(s))

THEOREM: single-l1
(bw-statep(s) $\wedge$ listp (cdr(s)))
$\rightarrow$ (result-new (move-over (last (car(s)), caadr(s)), s)
= cons (append (car(s), cadr(s)), cddr(s)))

Plan to form single tower. Measure is the number of stacks in s.

DEFINITION:

```
form-single-tower-gen (s)
=  if (¬ bw-statep (s)) ∨ (s ≃ nil)  then nil
   elseif cdr (s) ≃ nil  then nil
   else cons (move-over (last (car (s)), caadr (s)),
              form-single-tower-gen (result-new (move-over (last (car (s)),
                                                            caadr (s)),
                                                 s))) endif
```

DEFINITION:
planp-new (x)
= **if** $x \simeq$ **nil then t**
  else move-overp (car (x)) $\wedge$ planp-new (cdr (x)) **endif**

Some examples.

THEOREM: new-ex1
result-new (move-over (`'b`, `'d`), `'((a b c) (d e))`)
= `'((c) (a b d e))`

THEOREM: new-ex2
result-new (move-over (`'b`, `'e`), `'((a b c) (d e))`)
= `'(failed move-over b e ((a b c) (d e)))`

THEOREM: new-ex3
result-new (move-over (`'c`, `'d`), `'((a b c) (d e))`) = `'((a b c d e))`

THEOREM: new-ex4
resultlist-new (list (move-over (`'b`, `'d`), move-over (`'e`, `'c`)),
  `'((a b c) (d e))`)
= `'((a b d e c))`

THEOREM: mem-app1
$(x \notin y) \rightarrow ((x \in \text{append}(y, z)) = (x \in z))$

THEOREM: stackp-append
(stackp (*st1*) $\wedge$ stackp (*st2*) $\wedge$ disjoint (*st1*, *st2*))
$\rightarrow$ stackp (append (*st1*, *st2*))

THEOREM: disjoint-append
(disjoint (x, y) $\wedge$ disjoint (z, y)) $\rightarrow$ disjoint (append (x, z), y)

THEOREM: disjointlist-append
(stackp (x) $\wedge$ disjointlist (x, cons (v, w)) $\wedge$ bw-statep (cons (v, w)))
$\rightarrow$ disjointlist (append (x, v), w)

THEOREM: bws1
(bw-statep (s) $\wedge$ listp (cdr (s)))
$\rightarrow$ bw-statep (result-new (move-over (last (car (s)), caadr (s)), s))

form-single-tower-gen forms a single tower.

THEOREM: move-over-th1
(bw-statep (s) $\wedge$ listp (s))
$\rightarrow$ (cdr (resultlist-new (form-single-tower-gen (s), s)) $\simeq$ **nil**)

form-single-tower-gen generates valid plans.

THEOREM: isa-plan1
planp-new (form-single-tower-gen (s))

form-single-tower-gen results in a blocks world state.

THEOREM: move-over-bws1
(bw-statep (s) $\wedge$ listp (s))
$\rightarrow$ bw-statep (resultlist-new (form-single-tower-gen (s), s))

form-single-tower-gen works.

THEOREM: move-over-plan-works
```
(bw-statep (s)
 ∧  listp (s)
 ∧  (p1 = form-single-tower-gen (s))
 ∧  (s1 = resultlist-new (p1, s)))
→  (bw-statep (s1) ∧ planp-new (p1) ∧ (cdr (s1) ≃ nil))
```

A.3 Formalizing Plan Constraints

Here are examples and theorems mentioned in Chapter 4. The definitions were checked by the theorem prover. We first provide the definitions and theorems that can be proved automatically. Then we give the statements of the theorems that have not been checked.

Compute plan duration of p given the duration of move and unstack actions.

DEFINITION:
```
plan-duration (p, move-time, unstack-time)
=  if p ≃ nil then 0
   elseif movep (car (p))
   then move-time + plan-duration (cdr (p), move-time, unstack-time)
   else unstack-time + plan-duration (cdr (p),
                                      move-time,
                                      unstack-time) endif
```

Test plan-duration.

THEOREM: plan-dur1
plan-duration (`'((unstack a) (move b c) (move a b))`, 2, 1) = 5

Make-hist constructs the history generated by p in s.

DEFINITION:
make-hist $(s,\ p)$
= **if** $p \simeq$ **nil then** cons $(s,$ **nil**$)$
else cons $(s$, make-hist (result (car (p), $s)$, cdr $(p)))$ **endif**

Here is an example of what make-hist produces.

THEOREM: makeclear-ex2
makeclear-gen (`'c`, STATE1) = `'((unstack a) (unstack b))`

THEOREM: hist-ex1
make-hist (STATE1, makeclear-gen (`'c`, STATE1))

```
=  '(((a b c) (d e))
     ((b c) (a) (d e))
     ((c) (b) (a) (d e)))
```

The function other-stacks returns the list of stacks in a state s that do not contain the block b.

DEFINITION:
other-stacks $(b,\ s)$ = delete (find-stack-of-block $(b,\ s)$, $s)$

THEOREM: other-stacks-ex1
other-stacks (`'c`, STATE1) = `'((d e))`

Check-other-stacks takes a history h and a list of stacks l as arguments and checks if every state in h is a superset of l.

DEFINITION:
check-other-stacks $(l,\ h)$
= **if** $h \simeq$ **nil then t**
else subset $(l$, car $(h))$ $\wedge$ check-other-stacks $(l$, cdr $(h))$ **endif**

THEOREM: check-other-ex1
check-other-stacks (other-stacks (`'c`, STATE1),
make-hist (STATE1, makeclear-gen (`'c`, STATE1)))

Suppose we want a plan to clear a block and then keep it clear for n units of time. We can achieve this goal by clearing b and then executing the action (unstack b) n times in succession. The function unstackn generates a sequence of n actions of unstacking a block b.

DEFINITION:
```
unstackn ($b$, $n$)
=  if $n \simeq$ 0 then nil
   else cons (unstack ($b$), unstackn ($b$, $n - 1$)) endif
```

Make-clear-intn is the composite plan for achieving the goal of keeping b clear for at least n units of time after b becomes clear.

DEFINITION:
make-clear-intn (b, n, s) = append (makeclear-gen (b, s), unstackn (b, n))

THEOREM: clear-intn-ex1
make-clear-intn (`'c`, 2, STATE1)
= `'((unstack a) (unstack b) (unstack c) (unstack c))`

The predicate check-clearn checks if block b is clear for at least n units of time, i.e., in the the first n+1 states, of the history h.

DEFINITION:
```
check-clearn ($b$, $h$, $n$)
=  if $h \simeq$ nil then f
   elseif $n \simeq$ 0 then clear ($b$, car ($h$))
   else clear ($b$, car ($h$)) $\wedge$ check-clearn ($b$, cdr ($h$), $n - 1$) endif
```

The function exist-clear-state returns the subsequence of the history h starting with the earliest state in which b is clear in h to the end of h. If such a sequence does not exist in h then exist-clear-state returns F.

DEFINITION:
```
exist-clear-intn ($b$, $h$, $n$)
=  if $h \simeq$ nil then f
   else check-clearn ($b$, $h$, $n$) $\vee$ exist-clear-intn ($b$, cdr ($h$), $n$) endif
```

THEOREM: hist-ex2
make-hist (STATE1, make-clear-intn ('c, 2, STATE1))
```
= '(((a b c) (d e))
     ((b c) (a) (d e))
     ((c) (b) (a) (d e))
     ((c) (b) (a) (d e))
     ((c) (b) (a) (d e)))
```

DEFINITION:
H1 = make-hist (STATE1, make-clear-intn ('c, 2, STATE1))

THEOREM: ex-ex1
exist-clear-intn ('c, H1, 2)

Prove that the block being cleared is not moved.

DEFINITION:
```
number-of-times-moved ($p$, $b$)
=  if $p \simeq$ nil then 0
   elseif cadar ($p$) = $b$ then 1 + number-of-times-moved (cdr ($p$), $b$)
   else number-of-times-moved (cdr ($p$), $b$) endif
```

THEOREM: number-ex1
number-of-times-moved ('((unstack a) (unstack b)), 'c) = 0

THEOREM: number-ex2
number-of-times-moved ('((unstack a) (unstack b)), 'a) = 1

Check if every unstack action is immediately followed by a move action in a plan.

DEFINITION:
```
move-follows-unstackp ($p$)
=  if ($p \simeq$ nil) $\lor$ (cdr ($p$) $\simeq$ nil) then t
   elseif unstackp (car ($p$))
   then movep (cadr ($p$)) $\land$ move-follows-unstackp (cddr ($p$))
   else move-follows-unstackp (cdr ($p$)) endif
```

THEOREM: move-follows-ex1
$\neg$ move-follows-unstackp (makeclear-gen ('c, STATE1))

THEOREM: move-follows-ex2
move-follows-unstackp (`'((move a b) (unstack c) (move b c))`)

Check-both checks if the history generated by p clears b and that an (unstack b) action is executed in the state in which b becomes clear.

DEFINITION:
check-both $(h,\ p,\ b)$

```
= if h ≃ nil then f
  elseif clear (b, car (h)) then car (p) = unstack (b)
  else check-both (cdr (h), cdr (p), b) endif
```

DEFINITION:
P2 = append (makeclear-gen (`'c`, STATE1), list (unstack (`'c`)))

DEFINITION: H2 = make-hist (STATE1, P2)

THEOREM: check-both-ex1
check-both (H2, P2, `'c`)

The following have not been checked. The makeclear-gen program generates an optimal plan to clear a block. †

THEOREM: c4-th1
(bw-statep (s)
$\wedge$ find-stack-of-block $(b,\ s)$
$\wedge$ planp (p)
$\wedge$ $(s1 =$ resultlist $(p,\ s))$
$\wedge$ bw-statep $(s1)$
$\wedge$ clear $(b,\ s1))$
$\rightarrow$ (len (makeclear-gen $(b,\ s)) \leq$ len $(p))$

Makeclear-gen generates optimal plans even when move actions take more time than unstack actions.

THEOREM: c4-th2
(bw-statep (s)
$\wedge$ find-stack-of-block $(b,\ s)$
$\wedge$ planp (p)
$\wedge$ $(s1 =$ resultlist $(p,\ s))$
$\wedge$ bw-statep $(s1)$
$\wedge$ clear $(b,\ s1)$
$\wedge$ $(\textit{unstack-time} < \textit{move-time}))$
$\rightarrow$ (plan-duration (makeclear-gen $(b,\ s)$, *move-time*, *unstack-time*)
$\leq$ plan-duration $(p$, *move-time*, *unstack-time*$))$

Complexity of makeclear-gen - O(n) where n is the number of blocks in stack containing b in s.

Theorem: c4-com1
(bw-statep (s) $\land$ find-stack-of-block $(b,\ s)$)
$\rightarrow$ (len (makeclear-gen $(b,\ s)$) $\leq$ len (find-stack-of-block $(b,\ s)$))

If a block b exists in a state s then the plan makeclear-gen $(b,\ s)$ does not disturb the stacks in s that do not contain b during execution.

Theorem: c4-th3
(bw-statep (s) $\land$ find-stack-of-block $(b,\ s)$ $\land$ (p = makeclear-gen $(b,\ s)$))
$\rightarrow$ check-other-stacks (other-stacks $(b,\ s)$, make-hist $(s,\ p)$)

make-clear-intn $(b,\ n,\ s)$ achieves the goal of keeping b clear for an interval of n units of time in the future of the initial state s.

Theorem: clear-int-th1
(bw-statep (s)
$\land$ find-stack-of-block $(b,\ s)$
$\land$ $(n \in \mathbf{N})$
$\land$ (p = make-clear-intn $(b,\ n,\ s)$)
$\land$ (h = make-hist $(s,\ p)$))
$\rightarrow$ exist-clear-intn $(b,\ h,\ n)$

We can state that the plan specified by makeclear-gen does not move the block being cleared as follows.

Theorem: ch4-th5
(bw-statep (s) $\land$ find-stack-of-block $(b,\ s)$)
$\rightarrow$ (number-of-times-moved (makeclear-gen $(b,\ s)$, b) = 0)

In plans generated by makeclear-gen every unstack action is not immediately followed by a move action.

Theorem: ch4-th6
(bw-statep (s) $\land$ find-stack-of-block $(b,\ s)$)
$\rightarrow$ ($\neg$ move-follows-unstackp (makeclear-gen $(b,\ s)$))

Stating that a plan to unstack a block as soon as it becomes clear works.

THEOREM: ch4-th7
(bw-statep (s)
$\land$ find-stack-of-block (b, s)
$\land$ (p = append (makeclear-gen (b, s), list (unstack (b))))
$\land$ (h = make-hist (s, p)))
$\rightarrow$ check-both (h, p, b)

Appendix B

The Mutilated Checkerboard Problem

Here is a formalization of the mutilated checker board problem using Bob's representation. I later learnt that Herb Simon uses a similar trick in discussing a solution to a "Cube-brick problem". Our formalization allows us to prove by induction that every sequence of domino placements leads to a state in which the number of covered black and white squares are equal whereas in the mutilated board the number of black and white squares are not equal and hence cannot be covered. An interesting feature is the definition of the color predicates in terms of the representation of the squares. White squares are those whose coordinates sum up to an even number and black squares are those whose coordinates sum upto an odd number. We prove the theorem for a nxn board using mathematical induction.

REPRESENTATION:

We assume that the squares are labeled with coordinates with (0,0) being the upper left hand corner and (n,n) being the diagonally opposite square. Thus, there are n+1 rows and columns in the board. If n=7, we are talking about the usual 8x8 checkerboard with the rows and columns numbered from 0 through 7. More generally, when n is odd, the number of rows and columns is even and vice versa. We use [0..m,0..n] as our notation for a board whose rows are numbered from 0 through m and whose and columns are numbered from 0 through n.

A square is represented by a pair cons $(x,\ y)$ where x and y are its coordinates.

DEFINITION:
squarep (x) = (listp (x) $\wedge$ (car (x) $\in$ $\mathbf{N}$) $\wedge$ (cdr (x) $\in$ $\mathbf{N}$))

A domino is a pair of adjacent squares.

Two squares are adjacent if one of their coordinates is the same and the other coordinate differs by one. Our predicate is true provided the sum of the coordinates of the first argument is less than that of the second argument. Thus, our definition is asymmetric but this will do - there is no loss of generality.

DEFINITION:

adjp $(s1, s2)$
$= \quad (((\mathrm{car}(s1) = \mathrm{car}(s2)) \wedge ((1 + \mathrm{cdr}(s1)) = \mathrm{cdr}(s2)))$
$\quad \vee \quad ((\mathrm{cdr}(s1) = \mathrm{cdr}(s2)) \wedge ((1 + \mathrm{car}(s1)) = \mathrm{car}(s2))))$

A square x in a [0..n,0..n] board is a pair of coordinates each less than or equal to n.

DEFINITION:
squarenp $(x, n) = (\mathrm{squarep}(x) \wedge (\mathrm{car}(x) \leq n) \wedge (\mathrm{cdr}(x) \leq n))$

Dominos are those that necessarily fall within the board.

DEFINITION:
dominop (x, n)
$= \quad (\mathrm{squarenp}(\mathrm{car}(x), n) \wedge \mathrm{squarenp}(\mathrm{cdr}(x), n) \wedge \mathrm{adjp}(\mathrm{car}(x), \mathrm{cdr}(x)))$

DEFINITION:
square-listp (x)
$=$ **if** $x \simeq$ **nil then t**
else squarep (car (x)) $\wedge$ square-listp (cdr (x)) **endif**

A board state is a list of covered squares.

DEFINITION: board-statep (x) = square-listp (x)

Having defined the set of all possible states of all possible checkerboards let us move on to actions. There is exactly one action: place a domino. Preconditions: The squares of the domino should not be already covered. We don't have to check if the dominos fall within the board because the dominop definition takes care of it.

The state got as a result of placing a domino x on a board in state s is given by the following function. If the preconditions of the action are not satisfied then it returns a non-board-statep whose car is `'failed`. This "error" state is retained when other actions are performed. Thus, a legal domino placement action is one that results in a board state.

DEFINITION:
res-place (x, s)
$=$ **if** (car $(x) \in s$) $\vee$ (cdr $(x) \in s$) **then** list (`'failed`, `'place`, x, s)
else cons (car (x), cons (cdr (x), s)) **endif**

DEFINITION:
```
result (a, s)
=  if car (s) = 'failed then s
   else res-place (cadr (a), s) endif
```

The state got from s as a result of doing a list of actions.

DEFINITION:
```
resultlist (l, s)
=  if l ≃ nil then s
   else resultlist (cdr (l), result (car (l), s)) endif
```

A constructor for the action place a domino.

DEFINITION: place (x) = list ('place, x)

A predicate for placing a domino. Sufficient for the cadr to be a domino.

DEFINITION: placep (x, n) = dominop (cadr (x), n)

DEFINITION:
```
place-planp (x, n)
=  if x ≃ nil then t
   else placep (car (x), n) ∧ place-planp (cdr (x), n) endif
```

To express the theorem, we need a predicate for the state with all squares covered except the corner squares (0,0) and (n,n). We write functions to generate the set of all squares on the board and then assert that the set minus the two squares is the set of covered squares in the desired state.

Make the set of all squares in the mth row from columns 0 through n.

DEFINITION:
```
make-row (m, n)
=  if n ≃ 0 then list (cons (m, 0))
   else append (make-row (m, n - 1), list (cons (m, n))) endif
```

Make the set of squares in rows 0 through m and columns 0 through n.

DEFINITION:
```
make-all-rows (m, n)
=  if m ≃ 0 then make-row (0, n)
   else append (make-all-rows (m - 1, n), make-row (m, n)) endif
```

Delete first occurrence of x in l.

DEFINITION:
```
delete $(x,\ l)$
= if $l \simeq$ nil then $l$
  elseif $x = \mathrm{car}(l)$ then $\mathrm{cdr}(l)$
  else cons (car $(l)$, delete $(x,\ \mathrm{cdr}(l))$) endif
```

A mutilated [0..n,0..n] board includes all squares in rows 0 through n except the squares `'(0 . 0)` and `'(n . n)`.

DEFINITION:
```
mutilated-board $(n)$
= delete (cons $(n,\ n)$, delete ('(0 . 0), make-all-rows $(n,\ n)$))
```

DEFINITION:
```
set-equal $(l1,\ l2)$
= if $l1 \simeq$ nil then $l2 \simeq$ nil
  else (car $(l1) \in l2$) $\wedge$ set-equal (cdr $(l1)$, delete (car $(l1)$, $l2$)) endif
```

A state in which all squares of the mutilated [0..n,0..n] board are covered is given by the following predicate.

DEFINITION:
all-covered-except-cornerp $(x,\ n)$ = set-equal (x, mutilated-board (n))

A white square is one whose coordinates add up to an even number. Otherwise it is a black square.

DEFINITION: oddp $(x) = ((x \textbf{ mod } 2) = 1)$

DEFINITION: whitep $(x) = (((\mathrm{car}(x) + \mathrm{cdr}(x)) \textbf{ mod } 2) = 0)$

DEFINITION: blackp $(x) = (((\mathrm{car}(x) + \mathrm{cdr}(x)) \textbf{ mod } 2) = 1)$

Functions to compute the number of white and black squares.

DEFINITION:
```
nwhite $(x)$
= if $x \simeq$ nil then 0
  elseif whitep (car $(x)$) then 1 + nwhite (cdr $(x)$)
  else nwhite (cdr $(x)$) endif
```

DEFINITION:
nblack (x)

```
= if $x \simeq$ nil then 0
  elseif blackp (car $(x)$) then 1 + nblack (cdr $(x)$)
  else nblack (cdr $(x)$) endif
```

THE PROOF.

What is the impossibility argument? In the desired state the number of covered white squares is not equal to number of covered black squares. Every placement operation covers exactly one white square and one black square. Thus, all states reachable starting with a state in which there are no dominos on the board have an equal number of covered white and black squares. Ergo the desired state is unachievable.

First we show that make-all-rows (n, n) constructs the set of all squares of a [0..n,0..n] board has an equal number of white and black squares when n is odd. When n is even, the number of white squares = number of black squares + 1. We will do this by induction on n. For this we will prove by induction that if n is odd the number of white squares is equal to the number of black squares.

Some number theory.

THEOREM: t3
$(x \geq 2) \rightarrow ((1 + (x - 2)) = ((1 + x) - 2))$

Odd numbers succeed even numbers.

THEOREM: t1
$((n \textbf{ mod } 2) = 0) \rightarrow (((1 + n) \textbf{ mod } 2) = 1)$

THEOREM: plus1
$(m + (1 + n)) = (1 + (m + n))$

If a square is white then the square got by adding one to its second coordinate (i.e. one adjacent to it is black).

THEOREM: sq-wb1
whitep (cons (m, n)) $\rightarrow$ blackp (cons $(m, 1 + n)$)

Even numbers succeed odd numbers.

THEOREM: t4
$((n \textbf{ mod } 2) = 1) \rightarrow (((1 + n) \textbf{ mod } 2) = 0)$

The square next to a black square is white.

THEOREM: sq-wb2
$\text{blackp}(\text{cons}(m, n)) \rightarrow \text{whitep}(\text{cons}(m, 1 + n))$

THEOREM: odd-even1
$((n \textbf{ mod } 2) \neq 0) \rightarrow ((n \textbf{ mod } 2) = 1)$

To prove that the number of white squares equals the number of black squares in a row when n is odd. Base case: n = 1. If there are 2 columns nblack = nwhite in a row.

THEOREM: l1
$\text{make-row}(m, 1) = \text{list}(\text{cons}(m, 0), \text{cons}(m, 1))$

THEOREM: equal-bw-row-base0
$\text{nwhite}(\text{make-row}(m, 1)) = \text{nblack}(\text{make-row}(m, 1))$

THEOREM: t5
$((y \textbf{ mod } 2) \neq 1) \rightarrow (((1 + y) \textbf{ mod } 2) = 1)$

THEOREM: append-assoc
$\text{append}(\text{append}(l1, l2), l3) = \text{append}(l1, \text{append}(l2, l3))$

THEOREM: t6
$(n \geq 2)$
$\rightarrow \ (\text{make-row}(m, n)$
$\quad = \ \text{append}(\text{make-row}(m, n - 2), \text{list}(\text{cons}(m, n - 1), \text{cons}(m, n))))$

The number of white/black squares got when we append two lists is the sum of the number of white/black squares of the individual lists.

THEOREM: nwhite-append1
$\text{nwhite}(\text{append}(l1, l2)) = (\text{nwhite}(l1) + \text{nwhite}(l2))$

THEOREM: nblack-append1
$\text{nblack}(\text{append}(l1, l2)) = (\text{nblack}(l1) + \text{nblack}(l2))$

In each row of a chess board [0..m,0..n] where n is odd the number of black squares is equal to the number of white squares.

THEOREM: equal-bw-row1
$\text{oddp}(n) \rightarrow (\text{nwhite}(\text{make-row}(m, n)) = \text{nblack}(\text{make-row}(m, n)))$

In an entire [0..m,0..n] board, n odd, there are an equal number of white and black squares.

THEOREM: eq-bw-board1
$\text{oddp}(n) \rightarrow (\text{nwhite}(\text{make-all-rows}(m, n)) = \text{nblack}(\text{make-all-rows}(m, n)))$

Instantiating m as n in the above we get the following.

THEOREM: eq-bw-board
$\text{oddp}(n) \rightarrow (\text{nwhite}(\text{make-all-rows}(n, n)) = \text{nblack}(\text{make-all-rows}(n, n)))$

If the number of rows and the number of columns are odd then the number of white squares is 1 + number of black squares.

DEFINITION: $\text{evenp}(x) = ((x \textbf{ mod } 2) = 0)$

The number of white squares of an even numbered row with an odd number of columns is equal to the number of black squares + 1.

THEOREM: zero-ident
$(m + 0) = \text{fix}(m)$

THEOREM: white-one-plus-black
$(\text{evenp}(m) \land \text{evenp}(n))$
$\rightarrow \quad (\text{nwhite}(\text{make-row}(m, n)) = (1 + \text{nblack}(\text{make-row}(m, n))))$

THEOREM: black-one-plus-white
$(\text{oddp}(m) \land \text{evenp}(n))$
$\rightarrow \quad (\text{nblack}(\text{make-row}(m, n)) = (1 + \text{nwhite}(\text{make-row}(m, n))))$

THEOREM: l13
$\text{make-all-rows}(1, n) = \text{append}(\text{make-row}(0, n), \text{make-row}(1, n))$

THEOREM: l14
$\text{evenp}(n) \rightarrow (\text{nwhite}(\text{make-all-rows}(1, n)) = \text{nblack}(\text{make-all-rows}(1, n)))$

THEOREM: base2
$(\text{nwhite}(\text{make-row}(0, n)) + \text{nwhite}(\text{make-row}(1, n)))$
$= \quad (\text{nblack}(\text{make-row}(0, n)) + \text{nblack}(\text{make-row}(1, n)))$

THEOREM: l15
$(m > 0)$
$\rightarrow\ ((\text{nwhite}(\text{make-row}(m - 1,\ n)) + \text{nwhite}(\text{make-row}(m,\ n)))$
$\quad = \ (\text{nblack}(\text{make-row}(m - 1,\ n)) + \text{nblack}(\text{make-row}(m,\ n))))$

THEOREM: t61
$(m \geq 2)$
$\rightarrow\ (\text{make-all-rows}(m,\ n)$
$\quad = \ \text{append}(\text{make-all-rows}(m - 2,\ n),$
$\qquad\qquad \text{append}(\text{make-row}(m - 1,\ n),\ \text{make-row}(m,\ n))))$

In a [0..m,0..n] checkerboard in which m is odd, the number of black squares is equal to the number of white squares.

THEOREM: l16
$((x1 + y1 + z1) = (x1 + y2 + z2))$
$= \ ((y1 + z1) = (y2 + z2))$

THEOREM: eq-bw-board2
$\text{oddp}(m) \rightarrow (\text{nwhite}(\text{make-all-rows}(m,\ n)) = \text{nblack}(\text{make-all-rows}(m,\ n)))$

If the number of rows and columns of a checkerboard is odd then the number of white squares in the board is 1 greater than the number of black squares.

DEFINITION:
f1 (x)
$=$ **if** $x \simeq 0$ **then t**
 else f1 $(x - 1)$ **endif**

THEOREM: l17
$(\text{evenp}(m) \land (m > 0)) \rightarrow (((m - 1) \ \textbf{mod}\ 2) = 1)$

THEOREM: white-one-plus-black2
$(\text{evenp}(m) \land \text{evenp}(n))$
$\rightarrow\ (\text{nwhite}(\text{make-all-rows}(m,\ n)) = (1 + \text{nblack}(\text{make-all-rows}(m,\ n))))$

Having established that the number of white squares is equal to the number of black sqaures in a board with an even number of rows and columns and that the number of white squares is 1 greater than the number of black squares in a board with an odd number of rows and columns, we would like to show that in a mutilated nxn board the number of white squares is less than the number of black squares.

Theorem: t7
$(n \in \mathbf{N}) \rightarrow (\text{cons}\,(m,\ n) \in \text{make-row}\,(m,\ n))$

Theorem: t8
$(x \in \mathit{l2}) \rightarrow (x \in \text{append}\,(\mathit{l1},\ \mathit{l2}))$

Theorem: m1
$((n \in \mathbf{N}) \wedge (m \in \mathbf{N})) \rightarrow (\text{cons}\,(m,\ n) \in \text{make-all-rows}\,(m,\ n))$

If we delete a white square from a list then the number of white squares in the new list is less than that in the old list.

Theorem: white-delete
$((x \in l) \wedge \text{whitep}\,(x)) \rightarrow (\text{nwhite}\,(\text{delete}\,(x,\ l)) < \text{nwhite}\,(l))$

The (n,n) square is white.

Theorem: white1
$\text{whitep}\,(\text{cons}\,(n,\ n))$

Deleting a white square does not change the number of black squares.

Theorem: black-same1
$\text{whitep}\,(x) \rightarrow (\text{nblack}\,(\text{delete}\,(x,\ l)) = \text{nblack}\,(l))$

Deleting `'(0 . 0)` and cons $(n,\ n)$ does not change the number of black squares in a list since both of them are white.

Theorem: black-del1
$\text{nblack}\,(\text{delete}\,(\text{cons}\,(n,\ n),\ \text{delete}\,(\texttt{'(0 . 0)},\ x))) = \text{nblack}\,(x)$

Theorem: mut-l2
$\text{oddp}\,(n) \rightarrow (n \neq \texttt{0})$

Theorem: mem-del1
$((x \in l) \wedge (x \neq y)) \rightarrow (x \in \text{delete}\,(y,\ l))$

Theorem: m2
$\texttt{'(0 . 0)} \in \text{make-all-rows}\,(m,\ n)$

THEOREM: mut-l1
$\text{oddp}(n) \rightarrow (\text{cons}(n,\ n) \in \text{delete}(\texttt{'(0 . 0)},\ \text{make-all-rows}(n,\ n)))$

In a mutilated board with an even number of rows and columns the number of white squares is less than the number of black squares.

THEOREM: mut1
$(\text{oddp}(n) \land (x = \text{mutilated-board}(n))) \rightarrow (\text{nwhite}(x) < \text{nblack}(x))$

In a mutilated board with an odd number of rows and columns greater than 1, the number of white squares is less than the number of black squares.

THEOREM: mut2
$(\text{evenp}(n) \land (x = \text{mutilated-board}(n)) \land (n > 0))$
$\rightarrow\ (\text{nwhite}(x) < \text{nblack}(x))$

In any mutilated board the number of white squares is less than the number of black squares.

THEOREM: mut3
$((x = \text{mutilated-board}(n)) \land (n > 0)) \rightarrow (\text{nwhite}(x) < \text{nblack}(x))$

Just to show what a mutilated board looks like for n = 7.

THEOREM: mutboard7
mutilated-board (7)
```
=  '((0 . 1) (0 . 2) (0 . 3) (0 . 4) (0 . 5)
     (0 . 6) (0 . 7) (1 . 0) (1 . 1) (1 . 2)
     (1 . 3) (1 . 4) (1 . 5) (1 . 6) (1 . 7)
     (2 . 0) (2 . 1) (2 . 2) (2 . 3) (2 . 4)
     (2 . 5) (2 . 6) (2 . 7) (3 . 0) (3 . 1)
     (3 . 2) (3 . 3) (3 . 4) (3 . 5) (3 . 6)
     (3 . 7) (4 . 0) (4 . 1) (4 . 2) (4 . 3)
     (4 . 4) (4 . 5) (4 . 6) (4 . 7) (5 . 0)
     (5 . 1) (5 . 2) (5 . 3) (5 . 4) (5 . 5)
     (5 . 6) (5 . 7) (6 . 0) (6 . 1) (6 . 2)
     (6 . 3) (6 . 4) (6 . 5) (6 . 6) (6 . 7)
     (7 . 0) (7 . 1) (7 . 2) (7 . 3) (7 . 4)
     (7 . 5) (7 . 6))
```

Since we have defined all-covered-except-cornerp as a predicate that checks if all squares of a board except the corner squares are covered, we want to show that in all such states the number of white squares is less than the number of black squares.

Deleting a non-white element from a list of white squares does not alter the number of white squares.

THEOREM: white-same1
$(\neg\ \text{whitep}\,(x)) \rightarrow (\text{nwhite}\,(\text{delete}\,(x,\ l)) = \text{nwhite}\,(l))$

THEOREM: white-del1
$((x \in l) \wedge \text{whitep}\,(x)) \rightarrow (\text{nwhite}\,(l) = (1 + \text{nwhite}\,(\text{delete}\,(x,\ l))))$

If two sets of squares are equal then the number of white/black squares in them are equal.

THEOREM: nwhite-eq1
$\text{set-equal}\,(l1,\ l2) \rightarrow (\text{nwhite}\,(l1) = \text{nwhite}\,(l2))$

THEOREM: black-same2
$(\neg\ \text{blackp}\,(x)) \rightarrow (\text{nblack}\,(\text{delete}\,(x,\ l)) = \text{nblack}\,(l))$

THEOREM: black-del2
$((x \in l) \wedge \text{blackp}\,(x)) \rightarrow (\text{nblack}\,(l) = (1 + \text{nblack}\,(\text{delete}\,(x,\ l))))$

THEOREM: nblack-eq1
$\text{set-equal}\,(l1,\ l2) \rightarrow (\text{nblack}\,(l1) = \text{nblack}\,(l2))$

If all the squares except the corner ones are covered in a board state then the number of white squares covered is less than the number of covered black squares.

For odd boards.

THEOREM: unequal1
$$
\begin{array}{ll}
\multicolumn{2}{l}{(\text{evenp}\,(n)} \\
\wedge & (n > 0) \\
\wedge & \text{board-statep}\,(x) \\
\wedge & \text{all-covered-except-cornerp}\,(x,\ n)) \\
\rightarrow & (\text{nwhite}\,(x) < \text{nblack}\,(x))
\end{array}
$$

For even boards.

THEOREM: unequal2
$(\text{oddp}(n) \land \text{board-statep}(x) \land \text{all-covered-except-cornerp}(x,\ n))$
$\rightarrow \ (\text{nwhite}(x) < \text{nblack}(x))$

For all boards.

THEOREM: unequal3
$((n > 0) \land \text{board-statep}(x) \land \text{all-covered-except-cornerp}(x,\ n))$
$\rightarrow \ (\text{nwhite}(x) < \text{nblack}(x))$

Now we want to show that the number of covered white squares is equal to the number of covered black squares in every state that arises starting with an initial state when there are no dominos on the board.

If the sum of two numbers is even then adding one to one of the numbers will make the sum odd and vice versa.

THEOREM: t10
$((1 + w) + z) = (1 + (w + z))$

THEOREM: t11
$(x + 1 + y) = (1 + (x + y))$

A domino covers exactly one white square and one black square.

THEOREM: domino-white1
$\text{dominop}(x,\ n)$
$\rightarrow \ (\text{nwhite}(\text{cons}(\text{car}(x),\ \text{cons}(\text{cdr}(x),\ y))) = (1 + \text{nwhite}(y)))$

THEOREM: domino-black1
$\text{dominop}(x,\ n)$
$\rightarrow \ (\text{nblack}(\text{cons}(\text{car}(x),\ \text{cons}(\text{cdr}(x),\ y))) = (1 + \text{nblack}(y)))$

If I place a domino on a board state to achieve a new board state then that board state will have the two squares under the domino covered.

THEOREM: place-dom1
$(\text{board-statep}(s) \land \text{placep}(a,\ n) \land \text{board-statep}(\text{result}(a,\ s)))$
$\rightarrow \ (\text{result}(a,\ s) = \text{cons}(\text{caadr}(a),\ \text{cons}(\text{cdadr}(a),\ s)))$

If the number of covered black and white squares in a state are equal then they remain the same after a legal domino placement.

THEOREM: bw-equal2
(board-statep (s)
$\land$ (nwhite (s) = nblack (s))
$\land$ placep $(a,\ n)$
$\land$ board-statep (result $(a,\ s)$))
$\to$ (nwhite (result $(a,\ s)$) = nblack (result $(a,\ s)$))

We need the following lemmas to prove that: If the number of covered white and black squares are equal then they will remain the same after every sequence of actions or plan of placing dominos.

We must prove that executing a plan in an error state results in the error state.

THEOREM: failed-state1
(car (s) = `'failed`) $\to$ (car (resultlist $(p,\ s)$) = `'failed`)

THEOREM: res2
(board-statep (s) $\land$ placep $(a,\ n)$ $\land$ ($\neg$ board-statep (result $(a,\ s)$)))
$\to$ (car (result $(a,\ s)$) = `'failed`)

THEOREM: s1
(car (s) = `'failed`) $\to$ ($\neg$ board-statep (s))

THEOREM: res3
(board-statep (s)
$\land$ place-planp $(p,\ n)$
$\land$ listp (p)
$\land$ board-statep (resultlist $(p,\ s)$))
$\to$ board-statep (result (car (p), s))

The following function is for forcing the prover to do the induction I want.

DEFINITION:
foo $(s,\ p)$
= **if** $p \simeq$ **nil then nil**
  else foo (result (car (p), s), cdr (p)) **endif**

If the number of covered white and black squares is equal in a state and a sequence of legal domino placement operations is executed then the number of covered white and black squares would be equal in the resulting state.

THEOREM: bcequal1
(board-statep (s)
 $\wedge$ (nwhite (s) = nblack (s))
 $\wedge$ place-planp (p, n)
 $\wedge$ board-statep (resultlist (p, s)))
 $\rightarrow$ (nwhite (resultlist (p, s)) = nblack (resultlist (p, s)))

Starting with an initial state s of an [0..n,0..n] board, n greater than 0, in which there are no covered squares there is no sequence of actions that will result in a state in which all the squares except (0,0) and (n,n) are covered.

THEOREM: tough-nut
(place-planp (p, n)
 $\wedge$ ($n > 0$)
 $\wedge$ ($s1$ = resultlist (p, **nil**))
 $\wedge$ board-statep ($s1$))
 $\rightarrow$ ($\neg$ all-covered-except-cornerp ($s1$, n))

Examples that test theorems.

DEFINITION:
STATE1 = `'((3 . 4) (3 . 5) (4 . 6) (4 . 7))`

THEOREM: board-state-example1
board-statep (STATE1)

THEOREM: board-state-example2
board-statep (**nil**)

DEFINITION: DOMINO1 = `'((0 . 1) 0 . 2)`

DEFINITION: ILLEGAL-DOMINO = `'((0 . 2) 0 . 1)`

THEOREM: adjacency-example1
adjp (car (DOMINO1), cdr (DOMINO1))

THEOREM: adjacency-example2
$\neg$ adjp (car (ILLEGAL-DOMINO), cdr (ILLEGAL-DOMINO))

THEOREM: dominop-example1
$\neg$ dominop (ILLEGAL-DOMINO, 3)

THEOREM: dominop-example2
¬ dominop (DOMINO1, `1`)

THEOREM: dominop-example3
dominop (DOMINO1, `2`)

DEFINITION: ACTION1 = place (DOMINO1)

THEOREM: placep-example1
placep (ACTION1, 7)

THEOREM: placep-example2
¬ placep (ACTION1, `1`)

THEOREM: result-example1
result (ACTION1, STATE1)

```
= '((0 . 1) (0 . 2) (3 . 4) (3 . 5) (4 . 6) (4 . 7))
```

DEFINITION: STATE2 = result (ACTION1, STATE1)

THEOREM: error-state-example1
result (ACTION1, STATE2)

```
= '(failed place
    ((0 . 1) 0 . 2)
    ((0 . 1) (0 . 2) (3 . 4) (3 . 5) (4 . 6) (4 . 7)))
```

THEOREM: error-state-example2
result (ACTION1, result (ACTION1, STATE2))

```
= '(failed place
    ((0 . 1) 0 . 2)
    ((0 . 1) (0 . 2) (3 . 4) (3 . 5) (4 . 6) (4 . 7)))
```

DEFINITION: BAD-PLAN = list (ACTION1, ACTION1)

DEFINITION: ACTION2 = place (`'((4 . 1) 4 . 2)`)

DEFINITION: GOOD-PLAN = list (ACTION1, ACTION2)

THEOREM: plan-ex1
place-planp (GOOD-PLAN, 7)

THEOREM: plan-ex2
place-planp (BAD-PLAN, 7)

THEOREM: plan-ex3
resultlist (GOOD-PLAN, STATE1)
```
=  '((4 . 1)
     (4 . 2)
     (0 . 1)
     (0 . 2)
     (3 . 4)
     (3 . 5)
     (4 . 6)
     (4 . 7))
```

THEOREM: plan-ex4
resultlist (BAD-PLAN, STATE1)
```
=  '(failed place
     ((0 . 1) 0 . 2)
     ((0 . 1) (0 . 2) (3 . 4) (3 . 5) (4 . 6) (4 . 7)))
```

THEOREM: row5
make-row (5, 7)
```
=  '((5 . 0)
     (5 . 1)
     (5 . 2)
     (5 . 3)
     (5 . 4)
     (5 . 5)
     (5 . 6)
     (5 . 7))
```

THEOREM: twobytwo
make-all-rows (1, 1) = `'((0 . 0) (0 . 1) (1 . 0) (1 . 1))`

THEOREM: mutboard2
mutilated-board (1) = `'((0 . 1) (1 . 0))`

B.1 Some Plans for Tiling Boards

Plans to fill a row and fill a board. The definitions were admitted, the examples proved automatically by the prover but the general theorems were not checked.

Definition:
```
fill-row $(m, n)$
= if $n \simeq 0$ then nil
  elseif $n = 1$ then list (place (cons (cons $(m, 0)$, cons $(m, 1)$)))
  else append (fill-row $(m, n - 2)$,
               list (place (cons (cons $(m, n - 1)$, cons $(m, n)$)))) endif
```

Check if columns 0 thru n of row m is not covered.

Definition:
```
empty $(m, n, s)$
= if $n \simeq 0$ then cons $(m, 0) \notin s$
  else (cons $(m, n) \notin s$) $\wedge$ empty $(m, n - 1, s)$ endif
```

Check if columns 0 thru n of row m is covered.

Definition:
```
covered $(m, n, s)$
= if $n \simeq 0$ then cons $(m, 0) \in s$
  else (cons $(m, n) \in s$) $\wedge$ covered $(m, n - 1, s)$ endif
```

An example of what fill-row does.

Theorem: fill-row-ex0
```
fill-row (2, 3)
=  '((place ((2 . 0) 2 . 1)) (place ((2 . 2) 2 . 3)))
```

Theorem: fill-row-ex1
```
(($s0$ = '((1 . 2) (1 . 3)))
 $\wedge$  ($p1$ = fill-row (2, 3))
 $\wedge$  ($s1$ = resultlist ($p1$, $s0$)))
$\rightarrow$  (covered (2, 3, $s1$) $\wedge$ place-planp ($p1$, 3) $\wedge$ board-statep ($s1$))
```

Plan generator for filling rows 0 to m and columns 0 to n.

Definition:
```
fill-board $(m, n)$
= if $m \simeq 0$ then fill-row $(0, n)$
  else append (fill-board $(m - 1, n)$, fill-row $(m, n)$) endif
```

THEOREM: fill-board-ex0
fill-board (2, 3)

```
=  '((place ((0 . 0) 0 . 1))
     (place ((0 . 2) 0 . 3))
     (place ((1 . 0) 1 . 1))
     (place ((1 . 2) 1 . 3))
     (place ((2 . 0) 2 . 1))
     (place ((2 . 2) 2 . 3)))
```

THEOREM: fill-board-ex1
set-equal (make-all-rows (2, 3), resultlist (fill-board (2, 3), **nil**))

Unchecked theorems. †

THEOREM: fill-row-works
(board-statep (*s0*)
$\wedge$ oddp (n)
$\wedge$ ($m \in \mathbf{N}$)
$\wedge$ empty (m, n, *s0*)
$\wedge$ (*p1* = fill-row (m, n))
$\wedge$ (*s1* = resultlist (*p1*, *s0*)))
$\rightarrow$ (covered (m, n, *s1*) $\wedge$ place-planp (*p1*, n) $\wedge$ board-statep (*s1*))

THEOREM: fill-board-works
(oddp (n) $\wedge$ ($m \in \mathbf{N}$))
$\rightarrow$ set-equal (make-all-rows (m, n), resultlist (fill-board (m, n), **nil**))

Appendix C

A General Framework

GENERAL THEORY FOR MODELING VARIOUS DOMAINS. We have a single machine whose set of states and set of actions are a superset of those of any other domain. Thus, every domain can be thought of as being enacted in this general machine.

The modeling conventions are:

1. States can be modeled by any data structure other than (list 'failed ...) This error state is used to handle the execution of actions whose preconditions are not satisfied.

2. The effects of actions are specified by a program that can take any arguments with the initial state as the last argument. The action is represented by the lambda expression that stands for the function that specifies it. Actions (lambda expressions) are represented so that they can be evaluated in a given state using the Lisp interpreter eval$.

E.g. (defn res-move (x y s) ... compute new state ..)
(defn move (x y) (make-action 'res-move (list x y)))
The action term is: (list 'move (list 'quote x) (list 'quote y)) formed so that result can call the Lisp interpreter eval$ on (move x y) to get (res-move x y s).

DEFINITION:
nth (n, l)
$=$ **if** $n \simeq 0$ **then** car (l)
  else nth $(n - 1$, cdr $(l))$ **endif**

DEFINITION:
len (l)
$=$ **if** $l \simeq$ **nil** **then** 0
  else $1 +$ len (cdr (l)) **endif**

To obtain the nth argument of an action we may use the following function since they are represented as (list 'quote x).

DEFINITION: argn $(n, a) =$ cadr (nth (n, a))

Check if a is an action that is specified by the function whose symbol is at.

DEFINITION: check-action $(a,\ at) = (\text{car}\,(a) = at)$

Constructor for actions (lambda expressions).

DEFINITION:
make-arglist (al)
$=$ **if** $al \simeq$ **nil then nil**
else cons (list (`'quote`, car (al)), make-arglist (cdr (al))) **endif**

DEFINITION:
make-action $(type,\ arglist) = \text{cons}\,(type,\ \text{make-arglist}\,(arglist))$

Definition of result

DEFINITION:
result $(a,\ s)$
$=$ **if** car $(s) =$ **`'failed`** **then** s
else eval\$ (**t**, append (a, list (list (`'quote`, s))), **nil**) **endif**

The state got from s as a result of doing a list of actions.

DEFINITION:
resultlist $(l,\ s)$
$=$ **if** $l \simeq$ **nil then** s
else resultlist (cdr (l), result (car (l), s)) **endif**

We define holds the same way we defined result. Like make action we have a make-fluent.

DEFINITION:
make-fluent $(type,\ arglist) = \text{cons}\,(type,\ \text{make-arglist}\,(arglist))$

DEFINITION:
holds $(x,\ s)$ = eval\$ (**t**, append (x, list (list (`'quote`, s))), **nil**)

END OF GENERAL THEORY

C.1 Blocks World within the General Theory

Here is a formalization of the blocks world as part of the general framework. This differs from our previous blocks world formalization only in the definitions of move, unstack, movep and unstackp. Of course, the definitions of result and resultlist are borrowed from the general theory.

A block is a litatom.

DEFINITION: blockp (x) = litatom (x)

A stack is a non-empty list of distinct blocks with the first block clear and the last block on the table.

DEFINITION:

```
stackp $(l)$
=  **if** $l \simeq$ **nil then f**
   **elseif** cdr $(l)$ = **nil then** blockp (car $(l)$)
   **else** blockp (car $(l)$) $\wedge$ (car $(l)$ $\notin$ cdr $(l)$) $\wedge$ stackp (cdr $(l)$) **endif**
```

Two lists are disjoint if they do not have common members.

DEFINITION:

```
disjoint $(l1, l2)$
=  **if** $l1 \simeq$ **nil then t**
   **else** (car $(l1)$ $\notin$ $l2$) $\wedge$ disjoint (cdr $(l1)$, $l2$) **endif**
```

A given list is disjoint from each of the members of the given list of lists.

DEFINITION:

```
disjointlist $(s1, l1)$
=  **if** $l1 \simeq$ **nil then t**
   **else** disjoint ($s1$, car $(l1)$) $\wedge$ disjointlist ($s1$, cdr $(l1)$) **endif**
```

A state is a set of stacks possibly empty. No two stacks have any blocks in common, i.e., they are pairwise disjoint.

DEFINITION:

```
bw-statep $(x)$
=  **if** $x \simeq$ **nil**
   **then if** $x$ = **nil then t**
        **else f endif**
   **else** stackp (car $(x)$)
        $\wedge$ disjointlist (car $(x)$, cdr $(x)$)
        $\wedge$ bw-statep (cdr $(x)$) **endif**
```

Here are the predicates on blocks world states.

The predicate ontable. A block is on the table provided it is last on the list of some stack.

We define a function bottomp that checks if a block is the bottom of a given stack.

Definition:

```
bottomp ($b$, $st$)
= if $st \simeq$ nil then f
  elseif cdr ($st$) $\simeq$ nil then $b =$ car ($st$)
  else bottomp ($b$, cdr ($st$)) endif
```

Find the stack to which a block belongs. This can also be used to assert that a block exists.

Definition:

```
find-stack-of-block ($b$, $s$)
= if $s \simeq$ nil then f
  elseif $b \in$ car ($s$) then car ($s$)
  else find-stack-of-block ($b$, cdr ($s$)) endif
```

Definition: ontable (b, s) = bottomp (b, find-stack-of-block (b, s))

A block is on top of another block if they appear in sequence in a tower.

The function consecp checks if the given blocks appear in succession in the list of blocks constituting the stack.

Definition:

```
consecp ($b1$, $b2$, $st$)
= if $st \simeq$ nil then f
  elseif cdr ($st$) $\simeq$ nil then f
  elseif car ($st$) $= b1$ then $b2 =$ cadr ($st$)
  else consecp ($b1$, $b2$, cdr ($st$)) endif
```

Definition:
on ($b1$, $b2$, s) = consecp ($b1$, $b2$, find-stack-of-block ($b1$, s))

A block is clear if it is the topmost block of a tower.

Definition: clear (b, s) = assoc (b, s)

Let us go on to actions. We have two actions, move and unstack. Actions are lambda expressions whose representation is constructed using make-action. Action terms are recognized using check-action of the general theory.

Constructor for the move action.

DEFINITION: move $(b1, b2)$ = make-action (`'res-move`, list $(b1, b2)$)

Recognizer for the move action.

DEFINITION: movep (x) = check-action $(x,$ `'res-move`$)$

Constructor for unstack.

DEFINITION: unstack (b) = make-action (`'res-unstack`, list (b))

Recognizer for unstack.

DEFINITION: unstackp (x) = check-action $(x,$ `'res-unstack`$)$

A plan in the blocks world is a sequence of either move or unstack actions. We allow empty plans.

DEFINITION: actionp (x) = (movep (x) $\vee$ unstackp (x))

DEFINITION:

```
planp (x)
=  if x ≃ nil then t
   else actionp (car (x)) ∧ planp (cdr (x)) endif
```

The specifications of the move and unstack actions are given by res-move and res-unstack as before.

DEFINITION:

```
delete (x, l)
=  if l ≃ nil then l
   elseif x = car (l) then cdr (l)
   else cons (car (l), delete (x, cdr (l))) endif
```

DEFINITION:
exec-move $(b1, b2, s)$
$=$ **if** cdr (assoc $(b1, s)$) $\simeq$ **nil**
then cons (cons $(b1$, assoc $(b2, s))$,
delete (assoc $(b1, s)$, delete (assoc $(b2, s), s)))$
else cons (cons $(b1$, assoc $(b2, s))$,
cons (cdr (assoc $(b1, s))$,
delete (assoc $(b1, s)$, delete (assoc $(b2, s), s))))$ **endif**

DEFINITION:
res-move $(b1, b2, s)$
$=$ **if** $(b1 = b2) \vee (\neg$ clear $(b1, s)) \vee (\neg$ clear $(b2, s))$
then list (`'failed`, `'res-move`, s)
else exec-move $(b1, b2, s)$ **endif**

DEFINITION:
exec-unstack (b, s)
$=$ **if** cdr (assoc (b, s)) $\simeq$ **nil** **then** cons (list (b), delete (assoc $(b, s), s)$)
else cons (cdr (assoc (b, s)), cons (list (b), delete (assoc $(b, s), s$))) **endif**

DEFINITION:
res-unstack (b, s)
$=$ **if** $\neg$ clear (b, s) **then** list (`'failed`, `'unstack`, s)
else exec-unstack (b, s) **endif**

The following is the same sequence of events that we used to prove the correctness of the plan generating program makeclear-gen with the stand-alone formalization of the blocks world.

We will first prove that the result of executing legal move and unstack actions in a blocks world state is a legal blocks world state since the proof is quite intricate. Then we will prove that plans achieve goals.

Some rewrite rules for result.

THEOREM: result1
(bw-statep $(s) \wedge ((\neg$ clear $(b1, s)) \vee (b1 = b2) \vee (\neg$ clear $(b2, s))))$
$\rightarrow$ (result (move $(b1, b2), s) =$ list (`'failed`, `'res-move`, s))

THEOREM: result5
(bw-statep (s)
$\wedge$ clear $(b1, s)$
$\wedge$ clear $(b2, s)$

$\wedge$ $(b1 \neq b2)$
$\wedge$ (cdr (assoc ($b1$, s)) $\simeq$ **nil**))
$\rightarrow$ (result (move ($b1$, $b2$), s)
  $=$ cons (cons ($b1$, assoc ($b2$, s)),
    delete (assoc ($b1$, s), delete (assoc ($b2$, s), s))))

Theorem: result6
(bw-statep (s)
$\wedge$ clear ($b1$, s)
$\wedge$ clear ($b2$, s)
$\wedge$ $(b1 \neq b2)$
$\wedge$ listp (cdr (assoc ($b1$, s))))
$\rightarrow$ (result (move ($b1$, $b2$), s)
  $=$ cons (cons ($b1$, assoc ($b2$, s)),
    cons (cdr (assoc ($b1$, s)),
      delete (assoc ($b1$, s), delete (assoc ($b2$, s), s)))))

The value of result for an unstack action.

Theorem: result2
(bw-statep (s) $\wedge$ ($\neg$ clear (b, s)))
$\rightarrow$ (result (unstack (b), s) $=$ list (`'failed`, `'unstack`, s))

Theorem: result3
(bw-statep (s) $\wedge$ clear (b, s) $\wedge$ (cdr (assoc (b, s)) $\simeq$ **nil**))
$\rightarrow$ (result (unstack (b), s) $=$ cons (list (b), delete (assoc (b, s), s)))

Theorem: result4
(bw-statep (s) $\wedge$ clear (b, s) $\wedge$ listp (cdr (assoc (b, s))))
$\rightarrow$ (result (unstack (b), s)
  $=$ cons (cdr (assoc (b, s)), cons (list (b), delete (assoc (b, s), s))))

Now we can disable the definitions of result, unstack and move since we have all the info as rewrite rules.

Event: Disable result.

Event: Disable unstack.

Event: Disable move.

We want to prove that if the preconditions are satisfied then the result of doing a move or unstack action is always a blocks world state. The new blocks world state is formed by deleting stacks and adding new stacks to the state. Therefore, we want to prove that deleting two stacks from a set results in a set.

If s1 is disjoint with all the stacks in l1 then it is disjoint with all the stacks got by deleting x from l1.

THEOREM: disj2
disjointlist ($s1$, $l1$) $\rightarrow$ disjointlist ($s1$, delete (x, $l1$))

Deleting a member from a set of stacks leaves it as a set of stacks.

THEOREM: del-set1
bw-statep (s) $\rightarrow$ bw-statep (delete (x, s))

Deleting the stacks on which b1 and b2 are results in a set of stacks.

THEOREM: set-del-b1-b2
(bw-statep (s) $\land$ clear ($b1$, s) $\land$ clear ($b2$, s) $\land$ ($b1 \neq b2$))
$\rightarrow$ bw-statep (delete (assoc ($b1$, s), delete (assoc ($b2$, s), s)))

Now we want to prove that adding the new stacks results in a set of stacks.

First we establish that the lists to be added are stacks. Then we prove that they are disjoint from the rest of the stacks in the state got by deleting the original stacks.

THEOREM: assoc-stack1
(bw-statep (s) $\land$ assoc (b, s)) $\rightarrow$ stackp (assoc (b, s))

The cdr of a stack is a stack provided it is not empty.

THEOREM: stackp-cdrx1
(stackp (st) $\land$ listp (cdr (st))) $\rightarrow$ stackp (cdr (st))

THEOREM: litatom-stack1
(bw-statep (s) $\land$ ($\neg$ litatom (b))) $\rightarrow$ ($\neg$ assoc (b, s))

We want to prove that moving a block from st1 to the top of st2 results in a stack provided st1 is disjoint from st2. We do so by proving the following lemma. If there are two distinct stacks st1 and st2 in a state s and there is a block b in st1 then b is not in st2.

Disjointlist means disjoint with every member in the list.

THEOREM: disjoint-mem1
$(\text{disjoint}\,(s1,\ z) \wedge (x \in s1)) \rightarrow (x \notin z)$

THEOREM: disjointlist-mem1
$(\text{disjointlist}\,(s1,\ l) \wedge (s2 \in l) \wedge (x \in s1)) \rightarrow (x \notin s2)$

THEOREM: bex1
$(\text{bw-statep}\,(s)$
$\ \wedge\ \ (st1 \in s)$
$\ \wedge\ \ (st2 \in s)$
$\ \wedge\ \ (st1 \neq st2)$
$\ \wedge\ \ (b \in st1))$
$\rightarrow\ \ (b \notin st2)$

THEOREM: assoc-car1
$\text{assoc}\,(x,\ l) \rightarrow (\text{car}\,(\text{assoc}\,(x,\ l)) = x)$

Establish that b1 and b2 are on distinct stacks.

THEOREM: exec-movel1
$(\text{bw-statep}\,(s) \wedge \text{clear}\,(b1,\ s) \wedge \text{clear}\,(b2,\ s) \wedge (b1 \neq b2))$
$\rightarrow\ \ (\text{assoc}\,(b1,\ s) \neq \text{assoc}\,(b2,\ s))$

THEOREM: clear-stackp
$(\text{bw-statep}\,(s) \wedge \text{clear}\,(b,\ s)) \rightarrow (\text{assoc}\,(b,\ s) \in s)$

Seems to have trouble proving $\text{assoc}\,(b,\ s)$ is a listp.

THEOREM: stack-list1
$(\text{bw-statep}\,(s) \wedge \text{clear}\,(b,\ s)) \rightarrow \text{listp}\,(\text{assoc}\,(b,\ s))$

THEOREM: exec-movel2
$(\text{bw-statep}\,(s) \wedge \text{clear}\,(b1,\ s) \wedge \text{clear}\,(b2,\ s) \wedge (b1 \neq b2))$
$\rightarrow\ \ (b1 \notin \text{assoc}\,(b2,\ s))$

We can now establish that the new constructs are stacks.

THEOREM: new-stack1
$(\text{bw-statep}\,(s) \wedge \text{clear}\,(b1,\ s) \wedge \text{clear}\,(b2,\ s) \wedge (b1 \neq b2))$
$\rightarrow\ \ \text{stackp}\,(\text{cons}\,(b1,\ \text{assoc}\,(b2,\ s)))$

Now we want to prove that the new stacks are disjoint with the remaining stacks. We prove this by showing that: (a) If a stack is disjointlist with l1 then its cdr is disjointlist with it. (b) If two stacks are disjointlist with l1 then the stack formed by moving the top of one to the top of another is also disjointlist with l1.

If stacks st1 and st2 are disjoint with a stack st3 then the stack formed by moving the top of st1 to the top of st2 is also disjoint with st3.

THEOREM: disj-cons1
(stackp (*st1*)
 $\land$ stackp (*st2*)
 $\land$ stackp (*st3*)
 $\land$ disjoint (*st1*, *st3*)
 $\land$ disjoint (*st2*, *st3*))
$\rightarrow$ disjoint (cons (car (*st1*), *st2*), *st3*)

If there are two stacks that are disjoint with a list of stacks s then the stack formed by moving the top of one to the top of another is also disjointlist with s.

THEOREM: disjointlist-cons
(stackp (*st1*)
 $\land$ stackp (*st2*)
 $\land$ bw-statep (*s*)
 $\land$ disjointlist (*st1*, *s*)
 $\land$ disjointlist (*st2*, *s*))
$\rightarrow$ disjointlist (cons (car (*st1*), *st2*), *s*)

If a stack st1 is disjoint with another stack st2 then its cdr is also disjoint with it.

THEOREM: disjoint-cdr1
(stackp (*st1*) $\land$ stackp (*st2*) $\land$ disjoint (*st1*, *st2*))
$\rightarrow$ disjoint (cdr (*st1*), *st2*)

If a stack st1 is disjointlist with l1 then its cdr is also disjointlist with l1.

THEOREM: disjointlist-cdr
(stackp (*st1*) $\land$ bw-statep (*l1*) $\land$ disjointlist (*st1*, *l1*))
$\rightarrow$ disjointlist (cdr (*st1*), *l1*)

Basically we want to prove that a stack is disjoint from the rest of the members of a set of stacks: a key lemma.

The following are lemmas needed to prove that disjoint commutes.

THEOREM: disjoint-nlistp
$(s2 \simeq \textbf{nil}) \rightarrow \text{disjoint}(s1, s2)$

THEOREM: disjoint-cdr
$(\text{car}(s1) \notin s2) \rightarrow (\text{disjoint}(s2, \text{cdr}(s1)) = \text{disjoint}(s2, s1))$

Disjoint commutes.

THEOREM: disjoint-comm
$\text{disjoint}(s1, s2) = \text{disjoint}(s2, s1)$

Lemma: If a stack is deleted from a set of stacks then it is disjointlist with the remaining stacks.

THEOREM: del-disjointlist2
$(\text{bw-statep}(s) \land (st \in s)) \rightarrow \text{disjointlist}(st, \text{delete}(st, s))$

THEOREM: set-del3
$(\text{bw-statep}(s) \land (st1 \in s) \land (st2 \in s) \land (st1 \neq st2))$
$\rightarrow \quad (st1 \in \text{delete}(st2, s))$

Now we can prove that the stacks containing b1 and b2 are disjointlist with the set of stacks got by deleting them.

THEOREM: exec-movel5
$(\text{bw-statep}(s) \land \text{clear}(b1, s) \land \text{clear}(b2, s) \land (b1 \neq b2))$
$\rightarrow \quad \text{disjointlist}(\text{assoc}(b1, s), \text{delete}(\text{assoc}(b1, s), \text{delete}(\text{assoc}(b2, s), s)))$

THEOREM: exec-movel6
$(\text{bw-statep}(s)$
$\land \quad \text{clear}(b1, s)$
$\land \quad \text{clear}(b2, s)$
$\land \quad (b1 \neq b2)$
$\land \quad \text{listp}(\text{cdr}(\text{assoc}(b1, s))))$
$\rightarrow \quad \text{disjointlist}(\text{cdr}(\text{assoc}(b1, s)),$
$\qquad\qquad \text{delete}(\text{assoc}(b1, s), \text{delete}(\text{assoc}(b2, s), s)))$

We can prove one case when the rest of the stacks containing b1 is empty. When we add two stacks we must prove that the two are disjoint. These are formed from two existing stacks which we prove are disjoint.

The stacks on which two distinct clear blocks rest are disjoint.

THEOREM: set-disjoint1
(bw-statep ($s1$) $\wedge$ ($st1 \in s1$) $\wedge$ ($st2 \in s1$) $\wedge$ ($st1 \neq st2$))
$\rightarrow$ disjoint ($st1$, $st2$)

THEOREM: dist-stacks2
(bw-statep (s) $\wedge$ clear ($b1$, s) $\wedge$ clear ($b2$, s) $\wedge$ ($b1 \neq b2$))
$\rightarrow$ disjoint (assoc ($b1$, s), assoc ($b2$, s))

If two stacks st1 and st2 are disjoint then the stacks got by moving the top of st1 to the top of st2 are also disjoint.

THEOREM: disjoint-move1
(stackp ($st1$) $\wedge$ stackp ($st2$) $\wedge$ disjoint ($st1$, $st2$))
$\rightarrow$ disjoint (cons (car ($st1$), $st2$), cdr ($st1$))

THEOREM: disjoint-move2
(stackp ($st1$) $\wedge$ stackp ($st2$) $\wedge$ disjoint ($st1$, $st2$))
$\rightarrow$ disjoint (cdr ($st1$), cons (car ($st1$), $st2$))

THEOREM: exec-movel4
(bw-statep (s) $\wedge$ clear ($b1$, s) $\wedge$ clear ($b2$, s) $\wedge$ ($b1 \neq b2$))
$\rightarrow$ disjoint (cdr (assoc ($b1$, s)), cons ($b1$, assoc ($b2$, s)))

The stack formed by b1 on top of the stack assoc ($b2$, s) is disjoint with the remaining stacks.

THEOREM: exec-movel3
(bw-statep (s) $\wedge$ clear ($b1$, s) $\wedge$ clear ($b2$, s) $\wedge$ ($b1 \neq b2$))
$\rightarrow$ disjointlist (cons ($b1$, assoc ($b2$, s)),
delete (assoc ($b1$, s), delete (assoc ($b2$, s), s)))

If there is a state s in which there are distinct blocks b1 and b2 that are clear then moving block b1 to the top of b2 in s results in a blocks world state.

THEOREM: move01
(bw-statep (s)
$\wedge$ clear ($b1$, s)
$\wedge$ clear ($b2$, s)
$\wedge$ (cdr (assoc ($b1$, s)) $\simeq$ **nil**)
$\wedge$ ($b1 \neq b2$))
$\rightarrow$ bw-statep (result (move ($b1$, $b2$), s))

THEOREM: move02
(bw-statep (s)
$\land$ clear $(b1,\ s)$
$\land$ clear $(b2,\ s)$
$\land$ listp (cdr (assoc $(b1,\ s)$))
$\land$ $(b1 \neq b2)$)
$\rightarrow$ bw-statep (result (move $(b1,\ b2)$, s))

THEOREM: move-sit1
(bw-statep (s) $\land$ clear $(b1,\ s)$ $\land$ clear $(b2,\ s)$ $\land$ $(b1 \neq b2)$)
$\rightarrow$ bw-statep (result (move $(b1,\ b2)$, s))

We want to prove that unstack also results in a valid blocks world state.

THEOREM: unstackl1
(bw-statep (s) $\land$ clear $(b,\ s)$ $\land$ listp (cdr (assoc $(b,\ s)$)))
$\rightarrow$ ($b \notin$ cdr (assoc $(b,\ s)$))

THEOREM: unstackl2
(bw-statep (s) $\land$ clear $(b,\ s)$ $\land$ listp (cdr (assoc $(b,\ s)$)))
$\rightarrow$ disjointlist (cdr (assoc $(b,\ s)$), delete (assoc $(b,\ s)$, s))

We want to prove that if a stack is disjointlist with s then the stack formed by its car is also disjointlist with s. We prove the general theorem that if a stack is disjointlist with s then its subsets are also disjointlist with s.

DEFINITION:
subset $(s1,\ s2)$
= **if** $s1 \simeq$ **nil then t**
else (car $(s1) \in s2$) $\land$ subset (cdr $(s1)$, $s2$) **endif**

If a stack st1 is disjoint with a stack st2 then a subset of st1 is also disjoint with st2.

THEOREM: disj-subset
(stackp $(st1)$ $\land$ stackp $(st2)$ $\land$ subset $(st3,\ st1)$ $\land$ disjoint $(st1,\ st2)$)
$\rightarrow$ disjoint $(st3,\ st2)$

If there is a stack st that is disjoint with a set of stacks s then every subset of st is also disjoint with s.

THEOREM: disjointlist-subset
$(\text{stackp}(st1) \land \text{disjointlist}(st1, s) \land \text{bw-statep}(s) \land \text{subset}(st2, st1))$
$\rightarrow\ \text{disjointlist}(st2, s)$

THEOREM: disjointlist-single
$(\text{bw-statep}(s) \land \text{clear}(b, s))$
$\rightarrow\ \text{disjointlist}(\text{list}(b), \text{delete}(\text{assoc}(b, s), s))$

If a clear block is unstacked in a state then it results in a state.

THEOREM: unstack01
$(\text{bw-statep}(s) \land \text{clear}(b, s) \land (\text{cdr}(\text{assoc}(b, s)) \simeq \textbf{nil}))$
$\rightarrow\ \text{bw-statep}(\text{result}(\text{unstack}(b), s))$

THEOREM: unstack02
$(\text{bw-statep}(s) \land \text{clear}(b, s) \land \text{listp}(\text{cdr}(\text{assoc}(b, s))))$
$\rightarrow\ \text{bw-statep}(\text{result}(\text{unstack}(b), s))$

THEOREM: unstack-sit1
$(\text{bw-statep}(s) \land \text{clear}(b, s)) \rightarrow \text{bw-statep}(\text{result}(\text{unstack}(b), s))$

Having proved that the move and the unstack actions result in legal blocks world states, we turn our attention to verifying plans. Plans are expressed by plan generating Lisp programs and proved correct.

Example 1 : Plan to clear a block. The following lemmas are needed to establish admissibility of makeclear-gen. The measure len (find-stack-of-block (b, s)) decreases with every recursive call.

THEOREM: mem-find0
$(\text{bw-statep}(s) \land \text{find-stack-of-block}(b, s))$
$\rightarrow\ (\text{find-stack-of-block}(b, s) \in s)$

THEOREM: mem-find6
$(\text{bw-statep}(s) \land \text{find-stack-of-block}(b, s))$
$\rightarrow\ (b \in \text{find-stack-of-block}(b, s))$

THEOREM: find-mem-cdr
$(\text{bw-statep}(s)$
$\land\ \text{find-stack-of-block}(b, s)$
$\land\ (b \neq \text{car}(\text{find-stack-of-block}(b, s))))$
$\rightarrow\ (b \in \text{cdr}(\text{find-stack-of-block}(b, s)))$

Here is a theorem about the effect of unstack that we need. If I unstack the top of a non-empty stack in a state then the cdr of the stack is a member of the resulting state.

THEOREM: clear-car1
$(\text{bw-statep}\,(s) \wedge (st \in s)) \rightarrow \text{assoc}\,(\text{car}\,(st),\, s)$

THEOREM: disjoint-car1
$(\text{stackp}\,(st1) \wedge \text{stackp}\,(st2) \wedge \text{disjoint}\,(st1,\, st2))$
$\rightarrow \quad (\text{car}\,(st1) \neq \text{car}\,(st2))$

THEOREM: state-stackp
$(\text{bw-statep}\,(s) \wedge (\neg\ \text{stackp}\,(st))) \rightarrow (st \notin s)$

THEOREM: not-eq-car1
$(\text{bw-statep}\,(s) \wedge (st1 \in s) \wedge (st2 \in s) \wedge (st1 \neq st2))$
$\rightarrow \quad (\text{car}\,(st1) \neq \text{car}\,(st2))$

THEOREM: not-eq-car2
$(\text{bw-statep}\,(s) \wedge (st \in s) \wedge (st \neq \text{car}\,(s))) \rightarrow (\text{car}\,(st) \neq \text{caar}\,(s))$

THEOREM: state-cdr1
$(\text{listp}\,(s) \wedge \text{bw-statep}\,(s)) \rightarrow \text{bw-statep}\,(\text{cdr}\,(s))$

This is an important lemma. This states that if there is a stack st in s then trying to find the stack whose car is the top of st will result in st.

THEOREM: assoc-car2
$(\text{bw-statep}\,(s) \wedge (st \in s)) \rightarrow (\text{assoc}\,(\text{car}\,(st),\, s) = st)$

THEOREM: result7
$(\text{bw-statep}\,(s) \wedge (st \in s) \wedge \text{listp}\,(\text{cdr}\,(st)))$
$\rightarrow \quad (\text{result}\,(\text{unstack}\,(\text{car}\,(st)),\, s)$
$\qquad = \quad \text{cons}\,(\text{cdr}\,(st),\, \text{cons}\,(\text{list}\,(\text{car}\,(st)),\, \text{delete}\,(st,\, s))))$

THEOREM: unstack-eff1
$(\text{bw-statep}\,(s) \wedge (st \in s) \wedge \text{listp}\,(\text{cdr}\,(st)))$
$\rightarrow \quad (\text{cdr}\,(st) \in \text{result}\,(\text{unstack}\,(\text{car}\,(st)),\, s))$

EVENT: Disable bw-statep.

Proving find-assoc1

THEOREM: disjointlist-mem2
$(\text{disjointlist}\,(s1,\ l) \land (s2 \in l) \land (x \in s2)) \rightarrow (x \notin s1)$

THEOREM: find-stack3
$(\text{bw-statep}\,(s) \land (st \in s) \land (b \in st))$
$\rightarrow\ (\text{find-stack-of-block}\,(b,\ s) = st)$

If a block is clear then find-stack returns the same stack as does assoc. This is because we know b belongs to assoc $(b,\ s)$ and by mem-find3 it follows.

THEOREM: clear-mem1
$(\text{bw-statep}\,(s) \land \text{clear}\,(b,\ s)) \rightarrow (b \in \text{assoc}\,(b,\ s))$

THEOREM: find-assoc1
$(\text{bw-statep}\,(s) \land \text{clear}\,(b,\ s)) \rightarrow (\text{find-stack-of-block}\,(b,\ s) = \text{assoc}\,(b,\ s))$

THEOREM: find5
```
(bw-statep (s)
 ∧  find-stack-of-block (b, s)
 ∧  (b ≠ car (find-stack-of-block (b, s))))
 →  (find-stack-of-block (b,
                          result (unstack (car (find-stack-of-block (b, s))), s))
      =  cdr (find-stack-of-block (b, s)))
```

THEOREM: find-stack-listp
$(\text{bw-statep}\,(s) \land \text{find-stack-of-block}\,(b,\ s))$
$\rightarrow\ \text{listp}\,(\text{find-stack-of-block}\,(b,\ s))$

DEFINITION: $\text{m3}\,(b,\ s) = \text{len}\,(\text{find-stack-of-block}\,(b,\ s))$

Plan to clear a block by unstacking one by one the blocks on top of it.

DEFINITION:
```
makeclear-gen (b, s)
=  if (¬ find-stack-of-block (b, s)) ∨ (¬ bw-statep (s))  then f
   elseif b = car (find-stack-of-block (b, s))  then nil
   else cons (unstack (car (find-stack-of-block (b, s))),
              makeclear-gen (b,
                             result (unstack (car (find-stack-of-block (b,
                                                                        s))),
                                     s))) endif
```

Proving that plans generated by makeclear-gen achieve the desired goal.

THEOREM: mkc-l1
(bw-statep (s)
 $\land$ find-stack-of-block (b, s)
 $\land$ (car (find-stack-of-block (b, s)) = b))
$\rightarrow$ assoc (b, s)

THEOREM: mkc-l2
(bw-statep (s)
 $\land$ find-stack-of-block (b, s)
 $\land$ ($b \neq$ car (find-stack-of-block (b, s))))
$\rightarrow$ find-stack-of-block (b, result (unstack (car (find-stack-of-block (b, s))), s))

THEOREM: makeclear-works
(bw-statep (s) $\land$ find-stack-of-block (b, s))
$\rightarrow$ clear (b, resultlist (makeclear-gen (b, s), s))

Makeclear-gen generates valid plans.

THEOREM: make-clear-is-a-plan
planp (makeclear-gen (b, s))

Makeclear-gen results in a legal blocks world state.

THEOREM: makeclear-bwstate
(bw-statep (s) $\land$ find-stack-of-block (b, s))
$\rightarrow$ bw-statep (resultlist (makeclear-gen (b, s), s))

THEOREM: makeclear-works1
(bw-statep (s)
 $\land$ find-stack-of-block (b, s)
 $\land$ ($p1$ = makeclear-gen (b, s))
 $\land$ ($s1$ = resultlist ($p1$, s)))
$\rightarrow$ (bw-statep ($s1$) $\land$ planp ($p1$) $\land$ clear (b, $s1$))

Examples of theorems on concrete data and definitions used to express theorems that were not verified.

THEOREM: ch5-ex1
eval$ (`'list`, list (`'x`, `'y`), `'((x . 2) (y . 3))`) = `'(2 3)`

THEOREM: ch5-ex2
apply$ (`'plus`, `'(2 3)`) = `5`

THEOREM: ch5-ex3
eval$ (`t`, `'(plus x y)`, `'((x . 2) (y . 3))`) = `5`

THEOREM: ch5-ex4
eval$ (`t`, `'(plus (plus x y) (plus x y))`, `'((x . 2) (y . 3))`)
= `10`

DEFINITION:
append-template (*x*, *y*) = list (`'append`, list (`'quote`, *x*), list (`'quote`, *y*))

DEFINITION: STATE1 = `'((a b c) (d e))`

THEOREM: gen-ex1
result (list (`'res-move`, list (`'quote`, `'b1`), list (`'quote`, `'b2`)), STATE1)
= res-move (`'b1`, `'b2`, STATE1)

THEOREM: gen-ex2
argn (`1`, move (`'a`, `'b`)) = `'a`

DEFINITION: on-fluent (*b1*, *b2*) = make-fluent (`'on`, list (*b1*, *b2*))

DEFINITION:
find-stack-fluent (*b*) = make-fluent (`'find-stack-of-block`, list (*b*))

THEOREM: holds-ex1
holds (on-fluent (`'b`, `'c`), STATE1)

DEFINITION:
res-makeclear (*b*, *s*) = resultlist (makeclear-gen (*b*, *s*), *s*)

DEFINITION:
makeclear (*b*) = make-action (`'res-makeclear`, list (*b*))

THEOREM: complex-action-ex1
result (makeclear (`'b`), STATE1) = `'((b c) (a) (d e))`

DEFINITION:
makeclear-list-gen (l, s)
$=$ **if** $l \simeq$ **nil then nil**
  else cons (makeclear (car (l)),
        makeclear-list-gen (cdr (l), result (makeclear (car (l)), s))) **endif**

THEOREM: complex-action-ex2
makeclear-list-gen (`'(a b c)`, STATE1)

```
=  '((res-makeclear 'a)
     (res-makeclear 'b)
     (res-makeclear 'c))
```

THEOREM: complex-action-ex3
resultlist (makeclear-list-gen (`'(a b c)`, STATE1), STATE1)
$=$ `'((c) (b) (a) (d e))`

DEFINITION:
res-makeclear-list (l, s) = resultlist (makeclear-list-gen (l, s), s)

DEFINITION:
makeclear-list (l) = make-action (`'res-makeclear-list`, list (l))

THEOREM: complex-action-ex4
result (makeclear-list (`'(a b c)`), STATE1) = `'((c) (b) (a) (d e))`

Appendix D

Mechanization of $\mathcal{A}$

If we include domains as terms in the logic, then we can express facts describing the effects of actions in various domains without them interfering with each other. This obviates the need for non-monotonic reasoning. We define the interpreter resultlist with an extra domain parameter. The interpreter takes three arguments: a state which is an alist of fluent-value pairs where a value is 1 or 0, a domain represented as a set of e-propositions of A and a sequence of actions to be executed. It executes an action in a state according to the state transition function specified by the domain. Thus, the commonsense law of inertia is expressed as part of the definition of the result function.

Fluents.

DEFINITION: $\text{fluentp}\,(x) = \text{litatom}\,(x)$

Actions.

DEFINITION: $\text{actionp}\,(x) = \text{litatom}\,(x)$

A state is represented as list of fluent-value pairs. We represent values by 0 (false) or 1 (true). We assume that the first occurrence of a fluent in the state list gives its value so that we don't have to worry about duplicates in the list.

DEFINITION: $\text{valuep}\,(x) = ((x = \mathtt{0}) \lor (x = \mathtt{1}))$

DEFINITION:
```
fvlistp ($s$)
=  if $s \simeq$ nil then t
   else fluentp (caar ($s$)) $\land$ valuep (cdar ($s$)) $\land$ fvlistp (cdr ($s$)) endif
```

A-statep is a predicate on any state of any domain.

DEFINITION: $\text{a-statep}\,(x) = \text{fvlistp}\,(x)$

D-statep is a predicate on the states of a domain if the list of fluents of the domain is passed to it. has-fluents checks if every fluent in the fluent list occurs in the state.

DEFINITION:
has-fluents $(l,\ s)$
$=$ **if** $l \simeq$ **nil then t**
  else assoc (car (l), s) $\land$ has-fluents (cdr (l), s) **endif**

DEFINITION: d-statep $(l,\ s)$ $=$ (a-statep (s) $\land$ has-fluents $(l,\ s)$)

To represent domains, we must represent e-propositions. An e-proposition is represented as a list whose car is an action name, cadr is an effect which is a fluent expression and the rest are fluent expressions corresponding to its preconditions. Action names and fluent names are represented by litatoms and fluent expressions are fluent-value pairs. Thus, an e-prop is a list whose car is an action and whose cdr is an fvlist.

DEFINITION: e-prop (x) $=$ (actionp (car (x)) $\land$ fvlistp (cdr (x)))

A domain is a list of e-propositions.

DEFINITION:
domainp (x)
$=$ **if** $x \simeq$ **nil then t**
  else e-prop (car (x)) $\land$ domainp (cdr (x)) **endif**

The function result executes a given action according to the domain passed as a parameter. Go through the domain; whenever you find a rule about the given action then check if its preconditions are true in the given state and if so collect the fluent that is its effect.

To check preconditions we need holds and holds-all. Holds takes a fluent expression and a state and checks if it holds. Naturally it only looks at the first pair whose car matches the fluent in the given fluent expression.

DEFINITION: holds $(\mathit{fexp},\ s)$ $=$ (assoc (car (fexp), s) $=$ fexp)

Holds-all takes a list of fluent expressions and returns T if ALL of them hold and F otherwise.

DEFINITION:

```
holds-all ($x$, $s$)
=  if $x \simeq$ nil then t
   else (assoc (caar ($x$), $s$) = car ($x$)) $\wedge$ holds-all (cdr ($x$), $s$) endif
```

Compute-effects takes a state, an action and a domain and returns the list of pairs that are its effect when executed in the state. It implements the common sense law of inertia.

DEFINITION:

```
compute-effects ($a$, $s$, $dom$)
=  if $dom \simeq$ nil then nil
   elseif ($a$ = caar ($dom$)) $\wedge$ holds-all (cddar ($dom$), $s$)
   then cons (cadar ($dom$), compute-effects ($a$, $s$, cdr ($dom$)))
   else compute-effects ($a$, $s$, cdr ($dom$)) endif
```

The result of executing an action in a state depends on the current domain.

DEFINITION:

result (a, s, dom) = append (compute-effects (a, s, dom), s)

The interpreter.

DEFINITION:

```
resultlist ($l$, $s$, $dom$)
=  if $l \simeq$ nil then $s$
   else resultlist (cdr ($l$), result (car ($l$), $s$, $dom$), $dom$) endif
```

D.1 Theorems about $\mathcal{A}$ Domains

Some theorems about domains that can be described in A.

Example 1: The fragile object domain of A.

DEFINITION:

FO-DOMAIN

```
=  '((drop (holding . 0) (holding . 1))
     (drop (broken . 1) (holding . 1) (fragile . 1)))
```

If Holding, Fragile and not Broken are true in a state s0 of the Fragile Object Domain and Drop is executed in s0 then not Holding, Fragile and Broken become true.

DEFINITION: holding (s) = holds (`'(holding . 1)`, s)

DEFINITION: fragile (s) = holds (`'(fragile . 1)`, s)

DEFINITION: broken (s) = holds (`'(broken . 1)`, s)

DEFINITION:
frag-statep (s) = d-statep (`'(holding broken fragile)`, s)

THEOREM: frag-th1
(frag-statep (*s0*)
∧ holding (*s0*)
∧ fragile (*s0*)
∧ (¬ broken (*s0*))
∧ (*s1* = result (`'drop`, *s0*, FO-DOMAIN)))
→ ((¬ holding (*s1*)) ∧ fragile (*s1*) ∧ broken (*s1*))

Stated using resultlist.

THEOREM: frag-th1a
(frag-statep (*s0*)
∧ holding (*s0*)
∧ fragile (*s0*)
∧ (¬ broken (*s0*))
∧ (*s1* = resultlist (`'(drop)`, *s0*, FO-DOMAIN)))
→ ((¬ holding (*s1*)) ∧ fragile (*s1*) ∧ broken (*s1*))

If Holding and Not Broken are true in a state s0 of the Fragile Object Domain and Drop is executed in s0 then not Holding becomes true.

THEOREM: frag-th2
(frag-statep (*s0*) ∧ holding (*s0*) ∧ (¬ broken (*s0*)))
→ (¬ holding (result (`'drop`, *s0*, FO-DOMAIN)))

THEOREM: frag-th2a
(frag-statep (*s0*) ∧ holding (*s0*) ∧ (¬ broken (*s0*)))
→ (¬ holding (resultlist (`'(drop)`, *s0*, FO-DOMAIN)))

Example 2: Yale Shooting domain.

DEFINITION: yale-statep (s) = d-statep (`'(alive loaded)`, s)

DEFINITION:
YALE-DOMAIN
= `'((load (loaded . 1))`
`(shoot (alive . 0) (loaded . 1))`
`(shoot (loaded . 0)))`

If the gun is not loaded and Fred is alive initially then Fred will be dead in the state got after executing load, wait and shoot.

THEOREM: ysp1
(yale-statep (*s0*)
∧ holds (`'(loaded . 0)`, *s0*)
∧ holds (`'(alive . 1)`, *s0*))
→ holds (`'(alive . 0)`,
resultlist (`'(load wait shoot)`, *s0*, YALE-DOMAIN))

Shooting will unload the gun in any Yale Shooting state.

THEOREM: ysp2
yale-statep (*s0*)
→ (¬ holds (`'(loaded . 1)`, result (`'shoot`, *s0*, YALE-DOMAIN)))

Example 3: The Murder Mystery Domain. It is the same as the Yale domain except for v-propositions. Reasoning about the past. If Fred is alive in the initial state and is dead after shooting and waiting then the gun was loaded in the initial state.

THEOREM: mm1
(yale-statep (*s0*)
∧ holds (`'(alive . 1)`, *s0*)
∧ holds (`'(alive . 0)`, resultlist (`'(shoot wait)`, *s0*, YALE-DOMAIN)))
→ holds (`'(loaded . 1)`, *s0*)

Another inference: NOT ALIVE AFTER WAIT; SHOOT.

THEOREM: mm2
(yale-statep (*s0*)
∧ holds (`'(alive . 1)`, *s0*)
∧ holds (`'(alive . 0)`, resultlist (`'(shoot wait)`, *s0*, YALE-DOMAIN)))
→ holds (`'(alive . 0)`, resultlist (`'(wait shoot)`, *s0*, YALE-DOMAIN))

Example 4: Stolen Car domain. There are no actions explicitly specified. Prove that the given information is inconsistent.

DEFINITION: stolen (s) = holds (`'(stolen . 1)`, s)

THEOREM: st2
((¬ stolen ($s0$)) ∧ stolen (resultlist (`'(wait wait wait)`, $s0$, **nil**)))
→ **f**

Other extensions. Similar e-propositions are no problem at all. Let us add SHOOT CAUSES NOT ALIVE IF VERYNERVOUS to the Yale domain and prove the theorem that if the victim is very nervous then he will die after shoot occurs.

DEFINITION:
YALE-DOMAIN1

```
= '((load (loaded . 1))
     (shoot (alive . 0) (loaded . 1))
     (shoot (loaded . 0))
     (shoot (alive . 0) (verynervous . 1)))
```

THEOREM: ysp3
holds (`'(verynervous . 1)`, s)
→ holds (`'(alive . 0)`, result (`'shoot`, s, YALE-DOMAIN1))

"Non-monotonicity" is also not a problem. Consider this simple example. Suppose we have fluents Light1 and Light2 and the e-prop Switch1 causes Light1. Then we can prove the theorem that if Light2 is off then it will remain off after Switch1. After we add the e-prop SWITCH1 CAUSES LIGHT2 we can prove the theorem that Switch1 turns on Light2.

DEFINITION: SWITCH-DOM1 = `'((switch1 (light1 . 1)))`

THEOREM: l-th1
holds (`'(light2 . 0)`, s)
→ holds (`'(light2 . 0)`, result (`'switch1`, s, SWITCH-DOM1))

DEFINITION:
SWITCH-DOM2

```
= '((switch1 (light1 . 1)) (switch1 (light2 . 1)))
```

THEOREM: l-th2
holds (`'(light2 . 0)`, s)
$\rightarrow$ holds (`'(light2 . 1)`, result (`'switch1`, s, SWITCH-DOM2))

DEFINITION:
switch-dom2-statep (s) = d-statep (`'(light1 light2)`, s)

THEOREM: l-th3
(switch-dom2-statep (s) $\wedge$ holds (`'(light2 . 0)`, s))
$\rightarrow$ holds (`'(light2 . 1)`, result (`'switch1`, s, SWITCH-DOM2))

We can even express the above theorem without explicitly constructing the new domain.

THEOREM: l-th4
holds (`'(light2 . 0)`, s)
$\rightarrow$ holds (`'(light2 . 1)`,
    result (`'switch1`,
        s,
        cons (`'(switch1 (light2 . 1))`, SWITCH-DOM1)))

Proving properties of states. The following predicate says that p is a plan whose actions belong to the set of actions aset.

DEFINITION:
klp (p, *aset*)
= **if** $p \simeq$ **nil then** $p =$ **nil**
  else (car (p) $\in$ *aset*) $\wedge$ klp (cdr (p), *aset*) **endif**

Always-holds is an operator that says that the fluent expression fexp holds in every state got by executing plan p in s1 according to the set of e-propositions dom provided p is a plan consisting of actions in aset.

DEFINITION:
always-holds (*fexp*, p, *aset*, *s1*, *dom*)
= (klp (p, *aset*) $\rightarrow$ holds (*fexp*, resultlist (p, *s1*, *dom*)))

If Fred is dead in a situation then there is no Yale Shooting action that can revive him.

THEOREM: fred-dead2

```
 holds ('(alive . 0), s1)
 →  always-holds ('(alive . 0),
                  p,
                  '(load wait shoot),
                  s1,
                  YALE-DOMAIN)
```

Reiter's example. If an object is broken then it cannot be fixed by dropping it.

THEOREM: always-broken

```
 holds ('(broken . 1), s)
 →  always-holds ('(broken . 1), p, '(drop), s, FO-DOMAIN)
```

McCarthy's example. Addition of a new action can achieve a goal previously unachievable. Add repair that simply makes broken false.

DEFINITION:

```
FO-DOMAIN1
=  '((drop (holding . 0) (holding . 1))
     (drop (broken . 1) (holding . 1) (fragile . 1))
     (repair (broken . 0)))
```

Now we can prove that a broken object can be repaired.

THEOREM: try1

```
 holds ('(broken . 1), s)
 →  holds ('(broken . 0), result ('repair, s, FO-DOMAIN1))
```

D.2 Verification of a Planner for $\mathcal{A}$ Domains

The following sequence of events verifies a general purpose planner for domains that can be described in A. This shows how solutions to problems that arise in a class of domains can be expressed and verified.

Here is a general purpose "planning program" that generates a sequence of actions of length n to transform an initial state to a goal state depending on the domain passed to it as a parameter. Thus, the solution generated by the program is applicable to a class of domains. An action such as wait is not needed for generating a plan because any goal that can be achieved with it may be achieved without it.

DEFINITION:
```
len (l)
=  if $l \simeq$ nil then 0
   else 1 + len (cdr (l)) endif
```

DEFINITION:
```
collect-actions (dom)
=  if dom $\simeq$ nil then nil
   else cons (caar (dom), collect-actions (cdr (dom))) endif
```

Function to form all plans of length n from a given list of actions actlist. Given a plan p the following function returns the set of plans got by tagging on each action in the action list actlist to the beginning of p.

DEFINITION:
```
add-action-to-plan (p, actlist)
=  if actlist $\simeq$ nil then nil
   else cons (cons (car (actlist), p),
              add-action-to-plan (p, cdr (actlist))) endif
```

Given a list of all plans of length n-1 form a list of all plans of length n that can be got by tagging on actions in actlist at the beginning.

DEFINITION:
```
form-longer-plans (plist, actlist)
=  if plist $\simeq$ nil then nil
   else append (add-action-to-plan (car (plist), actlist),
                form-longer-plans (cdr (plist), actlist)) endif
```

Return a list of all plans of length n given the list of actions actlist

DEFINITION:
```
form-all-plans (actlist, n)
=  if $n \simeq 0$ then list (nil)
   else form-longer-plans (form-all-plans (actlist, $n - 1$), actlist) endif
```

A planner - a function that finds a plan of length n to achieve a goal given as a list of fluent expressions when executed in an initial state s. If there is none return F. The algorithm used here is brute-force search - form all plans of length n from the given set of e-propositions dom. Return the first one that satisfies the goal.

Get-plan either returns a plan in plist that satisfies the goal if executed in the initial state or returns F if none of them satisfy it.

DEFINITION:
get-plan (*plist*, *dom*, *goal*, *s*)
= **if** *plist* $\simeq$ **nil then f**
elseif holds-all (*goal*, resultlist (car (*plist*), *s*, *dom*)) **then** car (*plist*)
else get-plan (cdr (*plist*), *dom*, *goal*, *s*) **endif**

DEFINITION:
find-plan (*s*, *dom*, *goal*, *n*)
= get-plan (form-all-plans (collect-actions (*dom*), *n*), *dom*, *goal*, *s*)

We want to prove the "completeness" of the planner, i.e., if there is a plan to achieve a goal then the planner would find it.

If there is a plan p consisting of actions of domain dom then p is a member of form-all-plans of length p using the actions in dom.

THEOREM: add-action1
$(a \in actlist) \rightarrow$ (cons $(a, p) \in$ add-action-to-plan $(p, actlist)$)

THEOREM: app-mem
$(x \in l1) \rightarrow (x \in$ append $(l1, l2))$

THEOREM: form-longer1
$((a \in actlist) \wedge (p \in plist))$
$\rightarrow$ (cons $(a, p) \in$ form-longer-plans $(plist, actlist)$)

THEOREM: l1
klp $(p, x) \rightarrow (p \in$ form-all-plans $(x,$ len $(p)))$

THEOREM: get-plan1
$((p \in plist) \wedge$ holds-all (*goal*, resultlist (p, s, dom)))
$\rightarrow$ holds-all (*goal*, resultlist (get-plan (*plist*, *dom*, *goal*, *s*), *s*, *dom*))

THEOREM: planner-complete
(domainp (x)
$\wedge$ klp $(p,$ collect-actions $(x))$
$\wedge$ a-statep $(s0)$
$\wedge$ fvlistp (*goal*)
$\wedge$ holds-all (*goal*, resultlist $(p, s0, x)$))
$\rightarrow$ holds-all (*goal*, resultlist (find-plan $(s0, x,$ *goal*, len $(p))$, $s0$, x))

Here are the examples given in Chapter 6.

Examples of states with duplicate occurrence of fluents.

DEFINITION:
STATE1 = `'((holding . 1) (fragile . 1) (broken . 0))`

THEOREM: state-example1
a-statep (STATE1)

DEFINITION:
STATE2

```
=  '((holding . 1)
     (fragile . 1)
     (broken . 0)
     (broken . 1))
```

THEOREM: state-example2
a-statep (STATE2)

THEOREM: state-example3
frag-statep (STATE1) $\wedge$ frag-statep (STATE2)

THEOREM: eprop-example1
e-prop (`'(drop (broken . 1) (holding . 1) (fragile . 1))`)

A representation of a domain.

THEOREM: domainp-example1
domainp (FO-DOMAIN)

Holds examples to show it checks only the first pair.

THEOREM: holds-example1
holds (`'(broken . 0)`, STATE1)

THEOREM: holds-example2
holds (`'(broken . 0)`, STATE2)

Examples of compute-effects. The effect of drop when holding and fragile are both true is not holding and broken whereas if it just not holding if fragile is false.

THEOREM: compute-effects1
compute-effects (`'drop`, STATE1, FO-DOMAIN)
= `'((holding . 0) (broken . 1))`

THEOREM: compute-effects3
compute-effects (`'drop`, STATE2, FO-DOMAIN)
= `'((holding . 0) (broken . 1))`

DEFINITION:
STATE3 = `'((holding . 1) (broken . 0) (fragile . 0))`

THEOREM: compute-effects2
compute-effects (`'drop`, STATE3, FO-DOMAIN) = `'((holding . 0))`

Result examples.

THEOREM: result-example1
result (`'drop`, STATE1, FO-DOMAIN)

```
=  '((holding . 0)
     (broken . 1)
     (holding . 1)
     (fragile . 1)
     (broken . 0))
```

THEOREM: result-example3
result (`'drop`, STATE2, FO-DOMAIN)

```
=  '((holding . 0)
     (broken . 1)
     (holding . 1)
     (fragile . 1)
     (broken . 0)
     (broken . 1))
```

THEOREM: result-example2
result (`'drop`, STATE3, FO-DOMAIN)

```
=  '((holding . 0)
     (holding . 1)
     (broken . 0)
     (fragile . 0))
```

Resultlist examples.

THEOREM: resultlist-example1
resultlist (`'(drop drop)`, STATE1, FO-DOMAIN)

```
=  '((holding . 0)
      (broken . 1)
      (holding . 1)
      (fragile . 1)
      (broken . 0))
```

THEOREM: resultlist-example2
resultlist (`'(drop drop)`, STATE2, FO-DOMAIN)

```
=  '((holding . 0)
      (broken . 1)
      (holding . 1)
      (fragile . 1)
      (broken . 0)
      (broken . 1))
```

BIBLIOGRAPHY

[1] W. W. Agresti, editor. *New Paradigms for Software Development.* IEEE Computer Society Press, 1986.

[2] James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning.* Morgan-Kaufmann, San Mateo, CA, 1990.

[3] James F. Allen. *Towards a general theory of action and time*, pages 464–479. In Allen et al. [2], 1990.

[4] Andrew Baker. Nonmonotonic reasoning in the framework of situation calculus. *Artificial Intelligence*, 49:5–23, 1991.

[5] William Bevier, Warren Hunt, J Strother Moore, and William Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4), 1989.

[6] D. Bjorner, C. A. R. Hoare, and H. Langmaack, editors. *VDM'90: VDM and Z — Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1990.

[7] A. Borgida, J. Mylopoulos, and R. Reiter. "...And Nothing Else Changes": The frame problem in procedure specifications. In *Proc. Fifteenth Int'l Conf. on Software Engineering*, 1993.

[8] Robert S. Boyer, David M. Goldschlag, Matt Kaufmann, and J. Strother Moore. Functional instantiation in first-order logic. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[9] Robert S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, New York, 1979.

[10] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[11] Robert S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In *Proceedings of the Eleventh Conference on Automated Deduction.* LNCS 607, Springer-Verlag, 1992.

[12] Robert S. Boyer and Yuan Yu. A formal specification of some user mode instructions for the Motorola 68020. Technical Report TR-92-04, Computer Sciences Department, University of Texas at Austin, 1992.

[13] R. Brachman and H. Levesque, editors. *Readings in Knowledge Representation.* Morgan Kaufmann Publishers, Inc., San Mateo, Calif., 1985.

[14] R. M. Burstall and J. A. Goguen. *An Informal Introduction to Specifications using CLEAR*, pages 363–390. In Gehani and McGettrick [34], 1986.

[15] E.M. Clark, M. C. Browne, E. A. Emerson, and A. P. Sistla. Using temporal logic for automatic verification of finite state systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 3–26. Springer-Verlag Berlin Heidelberg, 1985.

[16] B Cohen. Justification of formal methods for system specification. *Software Engineering Journal*, pages 26–35, January 1989.

[17] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[18] D. Good, et al. Report on the language GYPSY version 2.0. Technical Report ICSCA-CMP-10, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1978.

[19] O-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, Orlando, Fla., 1972.

[20] A. M. Davis. *Software Requirements: Analysis and Specification.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[21] J. De Kleer. *An Assumption-based TMS*, pages 280–298. In Ginsberg [38], 1987.

[22] E. W. Dijkstra. *Notes on Structured Programming*, pages 1–82. In [19], 1972.

[23] E. W. Dijkstra. Why correctness must be a mathematical concern. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science.* Academic Press, London, 1981.

[24] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[25] J. Doyle. *A truth maintenance system*, pages 259–279. In Ginsberg [38], 1987.

[26] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. North-Holland Publishing Company,Amsterdam, 1990.

[27] H Enderton. *A Mathematical Introduction To Logic.* Academic Press, Inc., Orlando, FL, 1972.

[28] S Fahlman. A planning system for robot construction tasks. *Artificial Intelligence*, 5:1–49, 1974.

[29] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

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

[31] Art Flatau. *A Verified Implementation of an Applicative Language with Dynamic Storage Allocation.* PhD thesis, University of Texas at Austin, 1992.

[32] Mark S. Fox and Stephen F. Smith. *A Knowledge-based System for Factory Scheduling*, pages 336–360. In Allen et al. [2], 1990.

[33] S. J. Garland, J. V. Guttag, and J. J. Horning. Debugging Larch Shared Language specifications. *IEEE Transactions on Software Engineering*, pages 1044–1057, September 1990.

[34] N. Gehani and A.D. McGettrick, editors. *Software Specification Techniques.* Addison-Wesley, 1986.

[35] Michael Gelfond and Vladimir Lifschitz. Representing actions in extended logic programming. In Krzysztof Apt, editor, *Proc. Joint Int'l Conf. and Symp. on Logic Programming*, pages 559–573, 1992.

[36] Michael Gelfond, Vladimir Lifschitz, and Arkady Rabinov. What are the limitations of the situation calculus? In Robert Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 167–179. Kluwer Academic, Dordrecht, 1991.

[37] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, Los Altos, CA, 1987.

[38] Mathew L. Ginsberg, editor. *Readings in Nonmonotonic Reasoning.* Morgan-Kaufmann, Los Altos, CA, 1987.

[39] David Gries. *The Science of Programming.* Springer-Verlag, Berlin, 1981.

[40] J. V. Guttag and J. J. Horning. *Formal Specification as a Design Tool*, pages 187–209. In Gehani and McGettrick [34], 1986.

[41] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, DEC Systems Research Center, July 1985.

[42] S. Hanks and D. McDermont. Default reasoning, nonmonotonic logics and the frame problem. In *Proc. AAAI-86*, volume 1, pages 328–333, 1986.

[43] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[44] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag Berlin Heidelberg, 1985.

[45] David Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.

[46] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, pages 330–338, November 1989.

[47] P. J Hayes. A logic of actions. In D Michie and B Meltzer, editors, *Machine Intelligence*, volume 6, pages 495–520. Wiley, New York, 1971.

[48] Pat Hayes, January 1993. In the Second Symposium on Logical Formalizations of Commonsense Reasoning.

[49] C. A. R. Hoare. *Notes on Data Structuring*, pages 83–174. In [19], 1972.

[50] J. R. Hobbs and R. C. Moore, editors. *Formal Theories of the Commonsense World*. Ablex, Norwood, NJ, 1985.

[51] IEEE. *Proceedings of the Fifth International Workshop on Software Specification and Design*, 1989.

[52] C. B. Jones and P. A. Lindsay. A support system for formal reasoning: Requirements and status. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88: VDM — The Way Ahead*, Lecture Notes in Computer Science, pages 139–152. Springer-Verlag, Berlin, 1988.

[53] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. *Mural: A Formal Development Support System*. Springer-Verlag, London, U.K., 1991.

[54] Cliff B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall International, Inc., London, 1980.

[55] Henry Kautz. The logic of persistence. In *Proc. of AAAI-86*, pages 401–405, 1986.

[56] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, pages 32–43, January 1985.

[57] A. Klausner and T. E. Konchan. *Rapid Prototyping and Requirements Specification Using PDS*, pages 441–454. In Gehani and McGettrick [34], 1986.

[58] F. Kluzniak and S. Szpakowicz. *Extract from Prolog for Programmers*, pages 140–153. In Allen et al. [2], 1990.

[59] R. A. Kowalski. *Logic for Problem Solving.* North-Holland, New York, 1979.

[60] C. Lafontaine, Y. Ledru, and P.-Y. Schobbens. An experiment in formal software development. *Communications of the ACM*, 34(5):62–71, 1991.

[61] V. Lifschitz. Formal theories of action. In *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, Los Angeles, CA, 1987.

[62] T. A. Linden. *Representing Software Designs as Partially Developed Plans*, pages 603–626. In Lowry and McCartney [65], 1991.

[63] B. H. Liskov and V. Berzins. *An Appraisal of Program Specifications*, pages 3–25. In Gehani and McGettrick [34], 1986.

[64] B.H. Liskov and J.V. Guttag. *Abstraction and Specification in Program Development.* MIT press, Cambridge, MA, 1986.

[65] Michael R. Lowry and Robert D. McCartney, editors. *Automating Software Design.* AAAI Press, Menlo Park, CA, 1991.

[66] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science.* Academic Press, London, 1981.

[67] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1992.

[68] Z. Manna and R. Waldinger. *Studies in Automatic Programming Logic.* North-Holland, New York, 1977.

[69] Z. Manna and R. Waldinger. How to clear a block: plan formation in situational logic. In *Proceedings of the Eighth Conference on Automated Deduction.* LNCS 230, Springer-Verlag, 1986.

[70] Z. Manna and R. Waldinger. The deductive synthesis of Imperative LISP programs. In *Proc. of AAAI-87*, pages 155–160, 1987.

[71] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. Technical Report STAN-CS-92-1404, Department of Computer Science, Stanford University, January 1992.

[72] Zohar Manna and Richard Waldinger. How to clear a block: A theory of plans. *Journal of Automated Reasoning*, 3:343–377, 1987.

[73] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. of AAAI-91*, pages 634–639, 1991.

[74] John McCarthy. Open problems in the epistemology of common sense. In the collection of papers given in his AI course at UT, Austin, Fall 1987.

[75] John McCarthy. A tough nut for proof procedures. Stanford University A.I. memo no. 16, 1964.

[76] John McCarthy. *Programs with Common Sense*, chapter 7. In Minsky [91], 1968.

[77] John McCarthy. Circumscription–a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1-2):27–39, 1980.

[78] John McCarthy. Invited commentary. In Jorg Siekmann and Graham Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 157–158. Springer Verlag, Berlin Heidelberg, 1983.

[79] John McCarthy. *Epistemological Problems of Artificial Intelligence*, pages 23–30. In Brachman and Levesque [13], 1985.

[80] John McCarthy. Applications of circumscription to non-monotonic reasoning. *Artificial Intelligence*, 28(1):89–116, 1986.

[81] John McCarthy. Generality in artificial intelligence. In Robert L. Ashenhurst, editor, *ACM Turing Award Lectures: The First Twenty Years*. ACM Press, New York, New York, 1987.

[82] John McCarthy. Overcoming an unexpected obstacle. 1992.

[83] John McCarthy, March 1993. Personal communication.

[84] John McCarthy. History of circumscription. *Artificial Intelligence*, 59:23–26, 1993.

[85] John McCarthy. Notes on formalizing context. In *Working Papers of the Second Symposium on Logical Formalizations of Commonsense Reasoning*, 1993.

[86] John McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In D. Michie and B. Meltzer, editors, *Machine Intelligence*, volume 4. Edinburgh University Press, Edinburgh, Scotland, 1969.

[87] Drew McDermont. *Reasoning about plans*, pages 269–318. In Hobbs and Moore [50], 1985.

[88] Drew McDermott. A critique of pure reason. *Computational Intelligence*, 3:151–160, 1987.

[89] Drew McDermott. *A Temporal Logic for Reasoning about Processes and Plans*, pages 436–463. In Allen et al. [2], 1990.

[90] Drew McDermott. Robot planning. *AI Magazine*, 13(2):55–79, 1992.

[91] M. Minsky, editor. *Semantic Information Processing.* MIT Press, Cambridge, MA, 1968.

[92] M. Minsky. A framework for representing knowledge. In Brachman and Levesque [13], pages 245–262.

[93] M. Minsky. *The Society of Mind.* Simon and Schuster, Inc., New York, 1985.

[94] Steve Minton, Craig A. Knoblock, Daniel R. Kuokka, Yolanda Gil, Robert L. Joseph, and Jaime G. Carbonell. Prodigy 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, School of Computer Science, Carnegie Mellon University, May 1989.

[95] J Strother Moore. Piton: A verified assembly-level language. Technical Report CLI-22, Computational Logic, Inc., Austin, Tx, June 1988.

[96] M. Moriconi. *A Designer/Verifier Assistant*, pages 335–350. In Rich and Waters [105], 1986.

[97] M. Moriconi and T. C. Winkler. Approximate reasoning about the semantic effects of program changes. *IEEE Transactions on Software Engineering*, pages 980–992, September 1990.

[98] A. Newell. Limitations of the current stock of ideas about problem-solving. In *Proceedings of a Conference on Electronic Information Handling*, pages 195–208, 1965.

[99] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice-Hall, 1972.

[100] N. J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971.

[101] E. P. D. Pednault. *Toward a mathematical theory of plan synthesis.* PhD thesis, Stanford University, Department of Electrical Engineering, 1986.

[102] E. P. D. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4:356–372, 1988.

[103] J. Scott Penberthy and Daniel S. Weld. Temporal planning with constraints. In *Proceedings of the Spring Symposium on Planning, Stanford, CA.*, 1993.

[104] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 1992.

[105] Charles Rich and Richard C. Waters, editors. *Readings in Artificial Intelligence and Software Engineering.* Morgan-Kaufmann, Los Altos, CA, 1986.

[106] Charles Rich and Richard C. Waters. *The Programmer's Apprentice.* ACM Press, Reading, MA, 1990.

[107] J. A. Robinson. Formal and informal proofs. In Robert Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 267–282. Kluwer Academic, Dordrecht, 1991.

[108] Lenhart Schubert. Monotonic solution of the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In H.E. Kyburg, R. Loui, and G. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer, 1990.

[109] Douglas R. Smith. *KIDS—A Knowledge-Based Software Development System*, pages 483–514. In Lowry and McCartney [65], 1991.

[110] J. M. Spivey. *Understanding Z.* Cambridge University Press, Cambridge, 1988.

[111] Werner Stephan and Susanne Biundo. A new logical framework for deductive planning. In *Proc. of IJCAI-93.*

[112] G. J. Sussman. *A Computational Model of Skill Acquisition.* American Elsevier, New York, 1975.

[113] Austin Tate, James Hendler, and Mark Drummond. A review of AI planning techniques. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 26–50. Morgan Kaufmann, San Mate, CA, 1990.

[114] Steven A. Vere. *Planning in Time: Windows and Durations for Activities and Goals*, pages 297–318. In Allen et al. [2], 1990.

[115] R. J. Waldinger. Achieving several goals simultaneously. In B. L. Webber and N. J. Nilsson, editors, *Readings in Artificial Intelligence.* Morgan Kaufmann, Los Altos, CA, 1981.

[116] Richard Waldinger. The bomb in the toilet. *Computational Intelligence*, 3:220–221, 1987.

[117] Jeanette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, Septermber 1990.

[118] T. Winograd. A procedural model of language understanding. In R. Schank and K. Colby, editors, *Computer Models of Thought and Language.* W. H. Freeman, San Francisco, 1973.

[119] T. Winograd. *Frame Representations and the Declarative/Procedural Controversy*, pages 357–370. In Brachman and Levesque [13], 1985.

[120] J. C. P. Woodcock. Structuring specifications in Z. *Software Engineering Journal*, pages 51–66, January 1989.

[121] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications.* Prentice Hall, Inc., 1984.

[122] W. D. Young. A Mechanically Verified Code Generator. *Journal of Automated Reasoning*, 5(4), 1989.

Index

VITA

Sakthikumar Subramanian was born to Rajalakshmi and Ramamurthi Subramanian on March 6, 1962 in Pudukottai, Tamil Nadu, India. He received his B.Tech in Electrical Engineering from the Indian Institute of Technology, Madras, India, in 1983 and M.S. in Computer Science from the University of Tennessee, Knoxville, in 1984. He joined the Ph.D. program in Computer Science at the University of Texas, Austin, in 1985 where he was employed as a teaching assistant and research scientist at various times. He also served as a part-time lecturer at the Southwest Texas State University in Fall, 1992. He is married to Sandhya Sundaresan and has a son, Amrit.

Permanent address: 7205 Hart Ln # 3019
Austin, Texas 78731

This dissertation was typeset[1] with LaTeX by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin dissertation format was written by Khe-Sing The.