AN INVESTIGATION INTO THREE GAMES USING ISABELLE, A GENERIC THEOREM PROVER

by

MICHAEL (BEAU) PATRICK BURKE JR.

A THESIS

Presented to the Department of Computer and Information Science
and the Honors College of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Bachelor of Arts

June 2000

APPROVED: [redacted]

Dr. Amr Sabry, Department of Computer and Information Science

An Abstract of the Thesis of

Michael (Beau) Patrick Burke Jr. for the degree of Bachelor of Arts

in the Department of Computer and Information Science to be taken June 2000

Title: AN INVESTIGATION INTO THREE GAMES USING ISABELLE, A GENRIC THEOREM PROVER

Approved: [redacted]

Dr. Amr Sabry

Simple games are often deceptively so. Beneath the obvious layers of normal game play often lie distinct properties that allow a player to engage their opponent in such a way as to gain a mathematically provable advantage. The investigation of those properties, as well as others that assist in the utilization of them during game play, is the primary focus of research for this paper. Specifically investigated are a winning strateg for the game of tic-tac-toe, a winning strategy for the game of two-finger morra, and board symmetries in a simplified version of tic-tac-toe dubbed "ti-ta-to." Used in the investigation of these properties is a generic theorem proving software called Isabelle. Isabelle is a generic logical framework written in ML and based on a subset of higher-order logic called meta-logic.

ACKNOWLEDGEMENTS

The author owes a great debt to Professor Sabry for his attention and guidance during the preparation of this document. Without his profound insight, meticulous scrutiny, and unswerving patience, this work would not have reached the quality that it has. Additionally, a significant measure of gratitude is equally deserved by Professors Farley and Fracchia, who climbed on board at the last minute without a moment's hesitation. Their support won't soon be forgotten.

TABLE OF CONTENTS

APPENDIX

LIST OF FIGURES

INTRODUCTION

Why is tic-tac-toe a boring game? After all, it was fun way back in the days of elementary school. One could scratch out two perpendicular sets of parallel lines on any scrap of paper and expect to engage one's opponent for at least five minutes in the pursuit of a win. Sometimes one would win, and sometimes one would lose. What's for certain is that, at that young age, one wasn't quite certain how the game would turn out. After a month or two of play, however, those young minds begin to pick up on winning strategies. They learn that some moves are better than others, and that, by moving in a particular way, one can drastically increase one's chance of winning the game. Eventually, those strategies are also learned by one's opponent, and the game is slowly reduced to an endless series of draw after draw. No matter how well one picks one's moves the game always ends in a tie. Tic-tac-toe is a boring game because the vast majority of those who've played it have also figured out how to play optimally. It is boring because no one ever wins.

So why is it that no one ever wins? Such a question is far more interesting, from a mathematical viewpoint, than that of how fun the game is. Clearly there is some propert inherent in it that allows for this pattern of tie after tie to emerge. Buried within the lines, crosses, and circles lies a hidden rule that states "if one plays in a certain way, one can never lose." The discovery and formalization of that property, among others, is the motivation for this paper.

GAME THEORY

Game theory originated from the study of situations of conflict. In the 1940's, it gained widespread recognition in mathematical communities thanks to work done b Oskar Morgenstern and John Von Neumann, a brilliant mathematician who also made groundbreaking contributions to the world of computer science. The focus of their research was the application of game theory to the study of economic models [Sta99]. Since then, it has come to be used by mathematicians, political scientists, psychologists, and other researchers in a variety of research contexts.

The basic premise of game theory is that one or more participants are involved in an activity wherein each participant is attempting to make a claim on a limited amount of resources. The object of a game, in the most general of terms, is to be the participant with a claim to the highest valued resources [Ibid.]. For example, in a game of tic-tac-toe, the winner of the game is the player who has managed to claim three linearl adjoining sections of the playing field. In a game of chess, the initial situation is slightl different, as both players are already in possession of the maximum amount of resources. However, the goal of the game is still the same as the general description given above. Instead of having no resources at the onset of the game, each player starts with a specific amount of resources. The object of the game is then not to win by accumulating resources, but rather to deplete the resources of the other player. In the end the winner is still the player with the highest valued resources.

The science of winning a game is, therefore, the science of developing strategies that exploit the rules of the game in such a manner as to guarantee that a positive outcome is always reached. Ewald Burger, in his Introduction to the Theory of Games, defines a

strategy for a given game as "a complete plan of behavior which specifies the player's behavior, that is, his/[her] decisions, for all possible circumstances that may arise during the course of play" [Bur63]. An entirely winning strategy would be one in which a player is able, no matter what strategy their opponent happens to be using, to predict all of the potential outcomes resulting from an opponent' s decision. Consequently, that player can then play in such a manner as to make consistent progress towards the goal of winning. Finding those winning strategies, or some other properties about a game that can lead to the development of winning strategies, is often no easy task. Proving that they are correct is even less easy.

MOTIVATIONS FOR AUTOMATED REASONING

Anyone who has ever taken a year of college level calculus will probably agree to the assertion that writing formal mathematical proofs is often an arduous process. While there is no hard and fast reason why this is so, several aspects of proof authoring are likely culprits for the majority of trouble. For one, proofs and verifications of mathematically modeled systems often get bogged down in the consideration of multitudes of special cases. For a complete and accurate proof, each case must be shown to be true if the entire statement is to be accepted as true. It only takes one omission to render the entirety of one's work unsound. In addition, a proof most certainly relies on work done by another author. So, not only must proof authors concern themselves with validating their own statements, they must also ensure that the sources they are referencing have been validated themselves.

When considering that all human beings will make small, unobtrusive errors in logic or deduction at some point in their lives, often times at the very beginning of an important proof, it's no wonder that writing formal statements is a difficult task. What's worse is that these countless hours of work are often performed outside of an academic vacuum. Bugs in a computer controlled or computer-modeled system, when uncaught, are notoriously famous for causing extremely expensive and potentially life-threatening disasters. On July 22, 1962, all it took was a single error in one "minute detail of [a] program" to bring down a spacecraft carrying the Mariner I probe. Only 290 seconds after it was launched, the rocket returned to the ground as a ball of fire. One missing logical statement in one tiny section of computer code cost the United States, in this one instance, a whopping $18 million dollars [Set97]. If the code had been verified to be formally correct, the situation might have been avoided. One can only guess that the sheer magnitude of the software's size, and the corresponding chore of formally checking it, was too big to warrant spending the money on it. If the software designers had had a tool that made it easy to formalize their work, there's no doubt they would have used it.

LOGICAL FRAMEWORKS

Clearly a better system for managing and writing correct, formal proofs of mathematical systems is in order. If not merely to give tired researchers a break from exhausting hundreds of special cases, then to help prevent against disasters like that of the Mariner I.

In a most basic sense, a logical framework is just that. It is a better system for writing correct, formal proofs. In a more formal and accurate sense, though, a logical

framework is better described by Frank Pfenning in his essay, "The Practice of Logical Frameworks," as "a meta-language for the specification of deductive systems" [Pfe96]. For the uninitiated, the terminology within this description may be somewhat confusing. To clarify, a deductive system is a set of axioms, or statements universally acknowledged to be true, and rules of inference, or rules that can be used to make mathematically sound deductions. Furthermore, the "meta" prefix denotes that a logical framework is a language for describing a language. It is a means through which a proof author can specify any arbitrary set of axioms and rules of inference with which to formalize a mathematical system or model.

Once a deductive system has been specified within the logical framework, a proof author can then instruct the software to use it, along with existing rules of inference, to tr and reach a proof goal. If the rules and axiom specified within the deductive system aren't enough to formally come to one's desired conclusion, one need only "fill in the gaps," so to speak. This may be as simple a task as proving a trivial lemma and then applying it to the desired proof goal. But, it could also be as complex as defining an entire theory to apply to the desired proof goal.

Before proceeding any further, it is important to note here that no deductive system works by magic. A computer is only as good as the smartest person giving it instructions. Currently, deductive software can't make profound, automatic leaps in comprehension. To paraphrase Lawrence Paulson, if one can't do a proof on paper then a logical framework isn't going to be much help [Pau99]. It is completely incapable of actually "coming up" with anything new. Computer programs can only apply what the already know about a specific problem to the process of solving the problem. If one

wants to try and prove something about set theory, for example, and the deductive system in use knows nothing about what it takes to prove a set-theoretic notion, then one would have an extremely hard time using that software to say *anything* at all about set-theoretic ideas. One would have to explain set theory to the computer, in the language specified b the logical framework, before it could be of any use to set theorists.

At the same time as the above statement illustrates a major problem with all human-designed computer systems, it also pointedly demonstrates the beauty of a well-designed logical framework. If the software doesn't know anything about a specific deductive system, all one has to do is tell the software how to do it once. From then on, a proof author can reference elements of that deductive system as often as that author so chooses. Once the software is aware of the rules, it doesn't forget them. Additionally, within a generic logical framework a proof author can formalize any amount of deductive systems that they so choose. So, if one rule set isn't providing any positive results towards a specific proof goal, all one has to do is "switch rules" and look at the problem from another perspective.

ISABELLE

Generalities

Isabelle version 98-1, is one such implementation of a generic logical framework. It is also the primary tool used in the research portion of this thesis. Isabelle is a deductive system based on a subset of higher-order logic called meta-logic [Pau94]. Much like a meta-language is a language for expressing languages, a meta-logic is a logic for expressing other logics, which are referred to as "theories" within Isabelle. To gain an

appreciation for the diversity of theories which can be represented with Isabelle, consider Figure 1. It describes a small subset of the theories already represented within Isabelle, each of which is included with the standard distribution of the software.

1. *Classical Logic* - The classical set of logical statements developed b Aristotle and company two thousand years ago.
2. *Zermelo-Fraenkel Set Theory*- A deductive system based on the theor of sets.
3. *Constructive Type Theory* - A theory based on the definitions of certain values and the operations permissible over the set of those definitions.

Figure 1. A subset of the theories included with the standard distribution of Isabelle

Using these standard theories, or a theory specified by a user, Isabelle conducts its proof authoring under user direction. That is, it waits for the user to give it a proof goal and then, under the direction of the user, applies either standard or user-defined "tactics" towards the resolution of that goal. A tactic could be described as a methodology for writing a proof. Proof by induction, which should be familiar to anyone who has taken a course in calculus, is one such proof tactic included with the standard distribution of Isabelle.

Consider the following example situation. To prove that multiplication is associative (i.e. that (2*3)*5 = 2*(3*5)), one would find a theory within Isabelle that was capable of representing the logic of simple mathematical statements, writing one if none existed. Then, using that theory, one would give Isabelle the goal that one wanted to reach, in this instance that "(A*B)*C = A*(B*C)." From there the user would apply system or user defined tactics to the problem of resolving the goal.

If Isabelle can't resolve the goal in one step, then there exists certain ambiguities within the body of the proof that Isabelle needs further direction in order to resolve. These ambiguities are solved in the same manner as the primary goal, by applying tactics to them. For example, in the process of trying to prove that multiplication is associative, it might first be necessary to prove that multiplication is also commutative, i.e. that "A*B = B*A."

In an ideal world, all a user would have to do is give Isabelle a goal and wait for it to display "yes, that's true" or "no, that's false." But given that there are potentially an infinite number of theories that can be used to express a particular statement and that, within a single theory, there are infinitely many ways to combine logical assertions in the search for goal resolution, such a hope is quite impractical. Therefore it must be enough that, if Isabelle cannot immediately give a proof author a definite "yes" or "no," it can provide a versatile, powerful, and extensible library of methods for quickly and formall determining whether or not a statement is true or false.

"Versatile, powerful, and extensible" are hard concepts to measure without direct examples of what has been done with a particular system. Thankfully, there alread exists a rich set of applications which have utilized Isabelle and demonstrated that it is, indeed, all of the above. For example, Lawrence C. Paulson has used it to verif cryptographic protocols in his theory set "Auth." Another of Isabelle's primary authors, Tobias Nipkow, has also used it to verify the Quicksort sorting algorithm in his "Qsort" theory file. Both theories are included with the standard distribution of Isabelle.

Reviewing completed work is an excellent measure for determining whether a particular system is sufficient for performing a given task. However, one can never sa

for sure how powerful a system is unless one tries to extend the boundaries of what has been accomplished with a given system. In an effort to explore the possibilities of Isabelle, as well as formally demonstrate some interesting properties about the chosen research material, Isabelle has been used during this period of research to formalize, verify, and prove properties about three specific games. At first, this application ma seem somewhat trivial. After all, of what consequence are theories about games when NASA rockets are blowing up over the skies of Florida? The answer lies in the fact that any model is useful. For every avenue of research there exists an application of that research that the original researcher hadn't intended. Just as washing laundry has similarities to how modern microprocessors operate, this research into games might have unforeseen links to something else beyond the scope of this author's understanding. If nothing else, it serves as another example of how the boundaries of Isabelle's functionality can be extended further.

Interaction With Isabelle

The most basic method of interacting with Isabelle is through the interactive M compiler that Isabelle sits on top of. From there, an Isabelle user can give the software a goal to reach, optionally a specific theory to work within, and then a series of tactics which work towards resolving that goal.

Goals can be given to Isabelle in a variety of different ways. Of primary interest in this document, though, are those given with the keywords "Goal," "Goalw," and "goal." (Note the case distinction between the first and the last.) The first method is the most basic format. It does no manipulation on the goal itself and defines the goal within

the current theory context. The second does the same as the first, only it can be given a list of rewrite rules to immediately apply to the goal. Again, it is defined within the current theory context. The last form is similar to the first in that no rewriting is done, but differs in that it requires a theory context be given within which to define the goal. Additionally, it will strip all high-order assumptions from the goal, if any, and assign them to a variable if given. A "goalw" function for establishing goals also exists, but it is not used within this research. (Its function can probably be derived from the other three).

Isabelle reaches finished proof states by having the user apply tactics and previously defined rules to a specified goal. Isabelle has a rich set of tactics which can be applied to a specific goal, far too many to describe here. But the majority of them are also very tuned to a specific usage, and are not as useful in a general context. Of particular interest in this research are three sorts of tactics. They are tactics that perform resolution, tactics that perform instantiation of variables. and tactics that perform simplification. Resolution tactics, like "resolve_tac" and "assume_tac," use existing rules, passed in as arguments, to generate new forms of the goal being operated upon. They do so by unifying the subgoal being operated on with the rules passed in, if possible, and then inserting those unified subgoals in place of the original [Pau96]. Tactics that perform instantiation of variables, like "exhaust_tac" and "induct_tac," replace variable instances with more concrete and workable forms of those variables. Simplification tactics, like "asm_full_simp_tac" and "simp_tac," apply rewrite rules to the subgoal the are resolving and then attempt to simplify that subgoal by deleting inconsequential statements from within the subgoal. Often this means simply removing logicall equivalent statements from both sides of an expression.

Interaction with Isabelle at the ML-compiler level is powerful, though also somewhat limited. When doing so, one is restricted to using theories that have alread been defined and stored on disk. Additionally, no record is kept by the software of work done during an interactive session. In order to fully exploit the functionality of Isabelle, a user must venture into the realm of user-defined theories and theory scripts. User-defined theories are no different than the fundamental logics contained within the base distribution of Isabelle [Pau96]. They are built by creating an Isabelle theory file, with the extension ".thy," and a theory script file, which has the same name as the theory file, but has an ".ML" extension instead. Theory script files are generally nothing more than a series of proof scripts, each of which is composed of a goal statement and sequence of tactics applied to that goal. Essentially, it is merely an interactive session stored on disk. Consequently, they will not be discussed in great detail, as the most interesting part of an Isabelle theory is the actual theory file.

Isabelle theory files all follow the same format, described in Figure 2. The letter "T" represents the name of the theory, and must match the name of the file that contains the theory, sans the extension. The series "S" represents all the other theories upon which the theory depends. (Note: These are not necessary, if the theory is completely new and not based on any other work.) After the name and extension declarations comes the bod of the theory. The first three headings are where classes, types, and arities (overloadings of pre-existing types) are defined [Pau96]. Following that are all constant declarations. These can represent specific instantiations of a given type, or functions that operate on defined types. Consistent with ML, functions are always declared and defined in curried form.

```
T = S1 + . . . + Sn +
classes          class declarations
default          sort
types            type declarations and synonyms
arities          arity declarations
consts           constant declarations
rules            rule declarations
translations     translation declarations
defs             const/rule/translation definitions
end
ML               ML Code
```

Figure 2. Isabelle theory file structure. Adapted from [Pau94].

After constants have been declared at the head of the theory file, they must be defined if anything is to be done with them. Definitions occur after the "defs" heading. Typically, they begin with an identifier, which is often the name of the constant followed by the characters "_def," and always end with a string representing the associated M code. Simple type instantiations and non-recursive functions, or functions which do not reference themselves, are all defined by equating the name of the constant with the value that defines it.

Recursive functions, which are also declared as constants within the theory file, are divided into two categories; primitive-recursive functions and non-primitive-recursive functions. Primitive-recursive functions are those for which clear progress towards termination is made during each recursive call. For example, the function F described in Figure 3.I is primitive-recursive, as each recursive call to F calls it for a number one less than N, where N is a natural number. Non-primitive-recursive functions are those which do eventually reach recursive termination, but which also don't make clear progress toward termination for each recursive call. The function F' described in Figure 3.II,

known as the Collatz problem, is an example of a function that is not primitive-recursive. Progress towards completion is unclear due to the conditional "N mod 2 == 0" [Lag96].

```
I. Primitive Recursion

     F(0)=      0
     F(N)=      F( N-1 ) + 1

II. Non-Primitive Recursion

     F'(1)=     X
     F'(N)=     if (N mod 2) == 0
                then F'(N/2)
                else F'(3*N + 1)
```

Figure 3. Primitive-and non-primitive-recursive functions.

Primitive-recursive function definitions begin with the "primrec" keyword, after which follows the two recursive forms of the function, namely the termination form of the function and the recursive form of the function. Optionally, an identifier can be given to each form of the function. Usually, though, this is not necessary since proof tactics automatically reference them if needed.

Non-primitive-recursive functions are defined by the "recdef" keyword, a function "measure," and all the forms of the recursive function. The "measure" function takes a single argument, which is also the only argument passed to the recursive function, and must return a natural number. Moreover, that natural number result must be smaller and smaller for each successive recursive call. This function is used by the theory compiler to prove that the non-primitive-recursive function actually terminates and is safe to use in proofs (i.e. it won't recur infinitely during simplification).

TIC-TAC-TOE

Rules, Game Play, and Observations

Tic-tac-toe, a game most should be familiar with, is played on a three by three board by two distinct players typically denoted by the symbols "X" and "O." One of the players begins the game by placing a move anywhere on the game board. The next player then places a move in any empty place on the game board, where an empty space is one that does not contain a move by either player X or player O. The first player then places another move in another empty space. This sequence of move/counter-move continues until either a player wins the game, or both players have filled the board with moves and are unable to place another (a draw). A player wins by placing three moves directl adjacent to each other on the game board, either horizontally, vertically or diagonall (Figure 4).

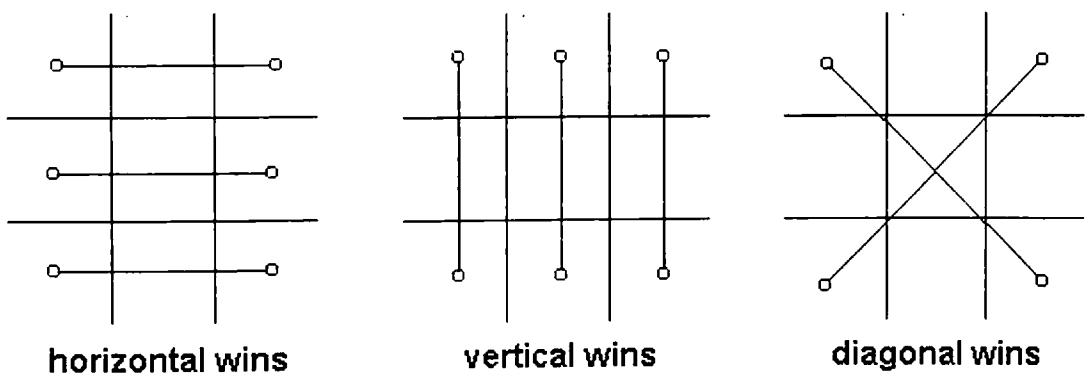

Figure 4. Winning combinations for tic-tac-toe.

Tic-tac-toe is a game where each player moves independently of the other player. That means, each player makes a move only after the other player has made a move, the

opening move being the only exception. It is also a game in which each round of pla depends on each preceding round of play. The manner in which each move changes the state of the board, from an undefined state to a winning state for example, is dependent on how all previous moves have been placed on the board. So, moves are not independent of each other. The significance of this fact will be discussed in great detail later in this paper, when the Isabelle implementation of tic-tac-toe is presented. However, at this point it suffices to say that there exists more than 300,000 different games of tic-tac-toe.

Goals for Research

Sorting through the 300,000 or so different games of tic-tac-toe would assuredl produce more than a few games in which one player had gained an advantage over the other. But, as was stated in the introduction, there seems to exist a way to play the game such that none of those games would ever result. In fact, if both players play using an optimal, winning strategy, then there really is no way for any of those games to emerge. Numerous strategies implement the optimal winning strategy, but for the purposes of this paper, only one has been chosen. That strategy is the mini-max strategy.

The mini-max strategy is not unique to tic-tac-toe and can, in fact, be applied to any game with a finite number of moves, complete visibility of all previous moves, and an alternating-turn style of game progression. The games of Reversi, also known as Othello, and Chess are both examples of such games.

The mini-max strategy works by "minimizing" the effect of one's opponent's moves on the state of the game and "maximizing" the effect of one's own moves. When it is the turn of the player using the mini-max strategy, that player evaluates how each

possible move at that point in the game would affect the state of the board. This evaluation is done by evaluating all the possible responses from that player's opponent to the move being evaluated. Naturally, in order to evaluate the opponent's responses, that mini-max employing player must evaluate how they would respond to their opponent's responses (Figure 5). When does all this evaluation stop? It stops when the evaluation has reached a full board, and no further responses can be generated.

Essentially, what the mini-max strategy is doing, is generating all possible games from a given starting point, namely the board at the time of the first player's turn. It then evaluates those games, determining whether they generate a win, loss, or tie, and picks the game with the best outcome.

Naturally, the algorithm for doing this is a little more involved than what is described above. But, it's not a whole lot more difficult than generating a tree, which represents all possible game "paths," and then doing a top-to-bottom traversal of the tree. During that traversal, one need only compare the states of each child node every time one visits a node with children. If the node represents a time when it is the player's turn, one then returns the better state up the tree. If that node represents a time when it is the player's opponent's turn, one returns the worse state up the tree. By doing so, one "maximizes" the player's return and "minimizes" the opponent's.

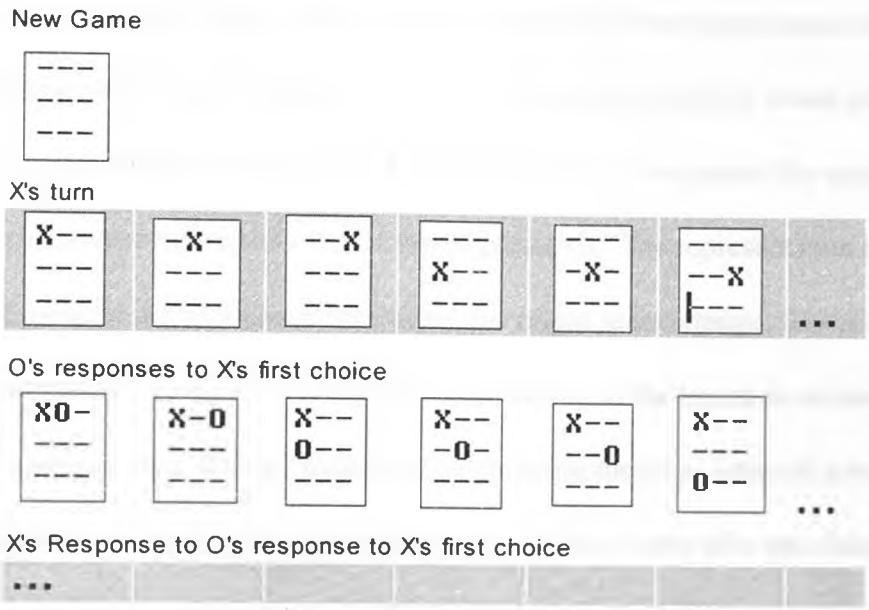

Figure 5. Top levels of a mini-max evaluation tree, starting with an empty game board. Only the first six nodes of the first level are shown, as well as the first six children of the first node on the first level

The mini-max strategy is implemented quite easily in almost any programming environment that supports recursive function calls, Isabelle included. Since Isabelle is able to understand the mini-max strategy, it is an ideal candidate for formal verification. Granted, it needs the support of a game board, players, and a way to make moves on that game board, but all those are all also equally achievable within Isabelle.

Implementation Notes

(Note: Please see appendix A for the full implementation of tic-tac-toe within Isabelle)

In order to do any work at all on the game of tic-tac-toe, the mini-max strateg needs primitives to operate on, as well as functions to operate on those primitives. To facilitate this need, an Isabelle theory file was developed which formalizes the minimum

functionality necessary to play a game of tic-tac-toe. Players are represented as a user defined datatype, moves are represented by a pair of natural numbers, board positions are represented as a datatype of either null (denoted "empty") or occupied (by a player), and the game board is represented as a list of board positions. The representation of a two-dimensional game board as a one-dimensional list might seem strange. However, as Professor Sabry noted early on in this project, the layout of the board is inconsequential with a little clever programming. Instead of referencing the (m,n) entry of a two-dimensional structure, one need only reference the (3*m,n) entry of a one-dimensional list to achieve an identical structure.

The majority of operations needed for tic-tac-toe are implemented as either non-recursive or primitive-recursive functions. Several key operations, notably the mini-max function, had to be implemented at non-primitive-recursive functions. This was done because they perform no destruction on their arguments, and consequently don't make clear progress towards termination. Admittedly, the mini-max function could very well have been written as two or more primitive-recursive functions. Unfortunately, due to time constraints a primitive-recursive solution could not be found. In addition, a brute force bound had to be placed on all non-primitive-recursive functions to convince the theory compiler that each terminates. This was done because the "measure" function for each before the bound was placed on it was not sufficient to prove that the function would terminate, as it merely called a function that did not formally produce a decreasing series of natural numbers.

Results

Formally proving the effectiveness of the mini-max strategy turned out to be a much harder problem to solve than originally expected. More than likely, a lot of the problems with the proof stemmed from the author's inexperience with Isabelle, formal proofs, and the functional programming paradigm. Significant amounts of time were invested in simply becoming familiar with the software and the concepts that it is founded on. The learning curve aside, though, there were a number of key issues with Isabelle and with the implementation of tic-tac-toe that led to the proof being abandoned in favor of more achievable goals.

For one, Isabelle seems to have difficulty resolving very large goals. The primar proof tactics used in the incomplete proof of the mini-max strategy relied heavily on term rewriting. Since the mini-max function recurs exponentially, the fairly simple goal given to Isabelle exploded into a statement of massive proportions. Exactly how big the statement was is inconsequential. However, a good guess as to the scope would factor in that there are over 300,000 different games of tic-tac-toe. Ordinarily, this is not an impossible number of cases to consider. A strictly ML version of the tic-tac-toe program, written as a precursor to the Isabelle version, reaches a draw state for the empty board in under five minutes on the author's seven year old machine. But such is not the case for the Isabelle version running on a top of the line workstation. Coming from a limited viewpoint, the author can only conclude that there must be something other than code execution going on in the background of Isabelle that makes such a large statement logistically unresolvable. Otherwise, all attempts to resolve that large would not have resulted in Isabelle crashing with an "Out of Memory" error.

It is a given that no decent computer scientist should ever be mastered by a large segment of code. So before giving up completely on the mini-max proof, an effort was made to try and reach the same goal in a smarter, more finessed, and more compact fashion. Again, though, all attempts at resolving the goal ran up against another difficult obstacle. Namely, the representation of the board did not lend itself very well to the proof goal. Generally, each proof attempt wound up in the same position, that of having to perform some sort of induction on the board itself. It would have been nice if that induction had been workable, as all theorems could have been proved for boards of size N by N, instead of just the three by three case. But, unfortunately, zero is included in the set of all numbers that N can represent. Not only does a board of size zero by zero make little sense in terms of a playable game, but all functions had been written without the empty list in mind. Trying to make a move on a zero-sized board, for example, would have resulted in a program error.

After much frustration with Isabelle, due to its limitations as well as those of the author, it was decided that a simpler game should be investigated, so that a better understanding of Isabelle could be developed before moving on to more difficult problems. Two-finger morra, a simple game of chance studied earlier in the term, was identified as a good test case and was explored with much more favorable results.

TWO FINGER MORRA

Rules, Game Play, and Observations

Two-finger morra, described in Saul Stahl's A Gentle Introduction to Game Theory, is a game of distinct and autonomous rounds played by two players. For each

round of play, both players hold up between zero and two fingers, as well as call out a number ranging from zero to four. The player that correctly calls out the sum of the number of fingers that both players are holding out wins that many points from the other. If both correctly guess the correct number of fingers, then neither player wins or loses an points. The same is true if neither of the players is able to correctly guess the number of fingers both have held up. The game ends when one player decides not to play any more. At that point, the winning player is the one who has the most points. Figure 6 describes three rounds of play for two-finger morra, as well as how each player's score changes throughout those rounds of play.

	Round N		Round N+1		Round N+2	
	Player 1	Player 2	Player 1	Player 2	Player 1	Player 2
Score	10	11	13	8	13	8
Hold	1	2	1	2	1	2
Call	3	2	1	1	3	3
Score Change	Add 3	Subtract 3	Add 0	Add 0	Add 0	Add 0

Figure 6. Three rounds of two-finger morra

Two-finger morra differs from tic-tac-toe in that each round of play is independent of all the rounds that preceded it. Granted, a player's score is entirely dependent on how well they do in each round, but the actual outcome of the game has nothing to do with preceding rounds. So, unlike tic-tac-toe with its over 300,000 different games, two-finger morra has only 225 different games (three different ways to hold out fingers times five different ways to call out a number, the product squared). It also differs in that each player is completely unaware of the other player's move until both have actually made

their move. These two facts might not seem significant, but they are essential to consider when developing a winning strategy. It should be plainly obvious that something like the mini-max strategy will not work. After all, it is dependent on the game having a finite number of moves and a player having complete knowledge of their opponent's moves. Consequently, another form of winning strategy must be developed, one that is not dependent on the state of the game or the eventual termination of the game.

Goals for Research

There is no surefire method for winning each and every round of two-finger morra. However, if one's opponent plays the game randomly, one can choose one's moves in such a way as to gain a distinct advantage in the long run of game play. By simply choosing to hold out one finger and call out three for each and every round of play, one is almost certain to walk away with a higher score than one's opponent.

Before attempting an explanation, it is important to note that this strategy, described in Saul Stahl's A Gentle Introduction to Game Theor , is entirely dependent on one's opponent playing completely randomly [Sta99]. Any variation in the opponent's strategy nullifies the probabilistic certainty upon which this winning strategy operates.

In order to understand how this strategy works, it should be observed that of the 225 possible games of two-finger morra, only fifteen need be considered. One's opponent still chooses from the full range of finger/call combinations (a total of fifteen). However, since one is only choosing to hold out one finger and call out three, the onl variation in the game results from one's opponent's moves. It is also essential to note

that, since one's opponent is playing completely randomly, each of those fifteen distinct games is equally likely to emerge at any point during game play.

Since each game is equally likely to occur, one can look at the outcome of an given round of play as the weighted average of the outcomes of all fifteen unique rounds of play. Begin by observing what occurs if one were to play the winning strategy against all fifteen possible combinations of fingers and calls, given in any order. At the end of those fifteen rounds, the player using the one/three combination would be nine points ahead (Figure 7). Using that sum and the fact that each of those games is equally likely, one can calculate that at end of each round of play, the player using the one/three combination will win 3/5's of a point (1/15 * 9 points) from their opponent. So, for each round of play, given that one's opponent is playing randomly, one always has a distinct advantage if one uses the winning strategy.

Player 1		Player 2		Score change
Hold	Call	Hold	Call	for Player
1	3	0	0	0
1	3	1	1	0
1	3	2	2	3
1	3	0	3	0
1	3	1	4	0
1	3	2	0	3
1	3	0	1	-1
1	3	1	2	-2
1	3	2	3	0
1	3	0	4	0
1	3	1	0	0
1	3	2	1	3
1	3	0	2	0
1	3	1	3	0
1	3	2	4	3
Player 1's score after all games				9

Figure 7. A hypothetical game of two-finger morra, in which each unique round of play is pitted against the winning strategy.

Again, it is important to note that this strategy only works if one's opponent is playing completely randomly, which is not very likely in a real game. The chances of this strategy working are even more unlikely if one's opponent catches on to what is happening, and begins to consistently hold out one finger and call out two. If this were to happen, not only could one not count on getting an average of 3/5's of a point for ever round played, one could most certainly count on *losing* two real points for each round that one sticks to holding out one finger and calling out three!

Implementation Notes

(Please see appendix C for the full implementation of two-finger morra)

Since nothing persists across rounds of two-finger morra aside from each player's score, a game is represented within Isabelle as a pair of natural numbers. Players also require no identification other than their score, so long as the order in which they are passed to the "playRound" function is preserved across the duration of the game. A move is expressed as a two element vector containing a "finger" datatype and a "call" datatype. There exists three constructors for the "finger" datatype, one for each possible number of fingers held up, and five constructors for the "call" datatype, one for each possible call. The decision to represent fingers and calls as a datatypes, rather than as natural numbers, was made early on to try and get away from potentially having to perform induction on them. In both cases, induction makes no sense whatsoever, since there is a distinct bound on how large each number can grow. However, variables of those types still might need to be instantiated in proofs. By representing them as datatypes, this worry is eliminated

via "exhaust_tac," which replaces instances of datatyped variables with all possible instantiations of that datatype.

Game play is expressed as a function called "playRound," which takes as arguments a game and two finger/call pairs. The function merely looks at the finger/call pairs for each player, adjusts scores accordingly, and returns the newly adjusted game. An additional function, "playNRounds" was written to facilitate playing a sequence of rounds. It takes a natural number argument, which represents the number of rounds to play, and an arbitrary game from which to begin score keeping. It then plays the winning combination against pseudo-random combinations of other legal moves for N rounds, where N is the natural number argument passed in.

Results

While not quite as spectacular as initially expected, the Isabelle proofs of the winning strategy for two-finger morra generated some very favorable results. A lot of groundwork had to be accomplished before they became feasible. In particular, a librar of theorems describing the less-than-or-equal-to operator was needed. Once those had been developed, however, a successful proof of the winning strategy, among others, was obtained (See appendix B for the full implementation of proofs relating to less-than-or-equal-to).

In an effort to probe the initial feasibility of proving the winning strategy for two-finger morra, individual proofs for each finger and call were performed. All of them were of the same form, that being "if the winning strategy is played against an opponent who holds out N fingers, the winning strategy will put one at an advantage no matter what

number the other player calls." Naturally, five of the proofs were of a similar nature, with the difference being that the call was the free variable and fingers was static. For rounds in which the winning strategy does not produce an immediate advantage, i.e. when one's opponent holds out one and calls two, an additional assumption had to be inserted into the goal in order for the proof to work. However, that was no more complicated than stating that one's opponent would not call or hold out the number that beats the winning strategy. Each was resolved by instantiating the free variable with all possible values for that datatype and then simplifying the resulting expressions.

Once those proofs had been accomplished, the winning strategy for an entire game of two-finger morra was formalized. Initially, the proof used the "playNRounds" function to play a series of fifteen rounds. However, the same problem of Isabelle's difficulty with large expressions was again encountered. In an effort to whittle down that expression into more manageable "chunks," all fifteen rounds were explicitly played in the body of the theorem and the sum of those results were analyzed with the less-than-or-equal-to operator. The results were positive. Since it suffices to show that the net increase in the score of the player using the winning strategy is greater than zero when played against all possible games, the proof was left at that state. Further work would incorporate the concept of a weighted average for each individual round. But, since that step is fairly inconsequential in the grand scheme of the proof, it has been omitted.

Now that real things had been proven about a real game, albeit a very simple one, another look at the game of tic-tac-toe was attempted.

TI-TA-TO

Rules, Game Play, and Observations

Ti-ta-to is a slight variation on classic tic-tac-toe, and was adopted from a course lecture on artificial intelligence given by Robin Burke at the University of Chicago [Bur97]. The only differences between ti-ta-to and tic-tac-toe are the boards' sizes and the number and types of winning combinations. Game play still consists of each player taking alternating turns and marking empty spaces with either an "X" or an "O," depending on which they've chosen to represent themselves with at the start of the game. In this version, though, both players make their moves on a two by two board. A player wins only by placing two moves adjacent to each other diagonally (Figure 8). As with tic-tac-toe, a draw is declared when all board spaces are marked with an "X" or and "O" and neither player has won.

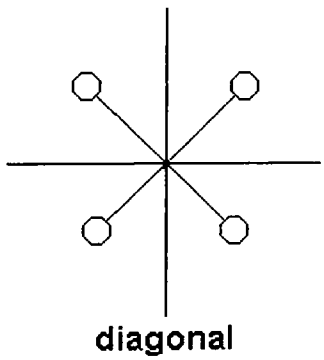

Figure 8. Winning combinations for ti-ta-to

Naturally, this game was chosen for investigation because of its similarities with tic-tac-toe, as well as the smaller amount of possible games. By cutting down the search

space so drastically, proofs about the game were much smaller and, consequently, much more manageable than those for tic-tac-toe

Goals

While a proof of the mini-max strategy remains a noble and achievable goal, further investigation into it was set aside in favor a different task. In its place, an investigation into the symmetry of the ti-ta-to game board was established. In this case "symmetry" actually refers to a rotation of the game board rather than a "mirroring" of it. But the idea is very much the same. Essentially, what is to be proved is that, no matter how the game board is oriented with respect to a given player, any rotations of the game board will have no effect on the state of the game.

Such an observation may seem somewhat trivial at first glance. But the verification of said property does have significant implications. By verifying that the ti-ta-to board is symmetric and then applying that fact to the development of a mini-max algorithm, or any algorithm that searches through a tree of all possible games, one can significantly speed up the performance of the mini-max algorithm. One need only find symmetrical paths within the tree and eliminate them if they have already been searched. In the case of ti-ta-to, for example, a game progress tree starting from the empty board would have four children at the level immediately following the root node. However, careful observation reveals that each of those moves is symmetric, so only one of those paths need be explored. With only one observation, the search space that the algorithm has to explore has been cut down by a factor of three fourths (Figure 9).

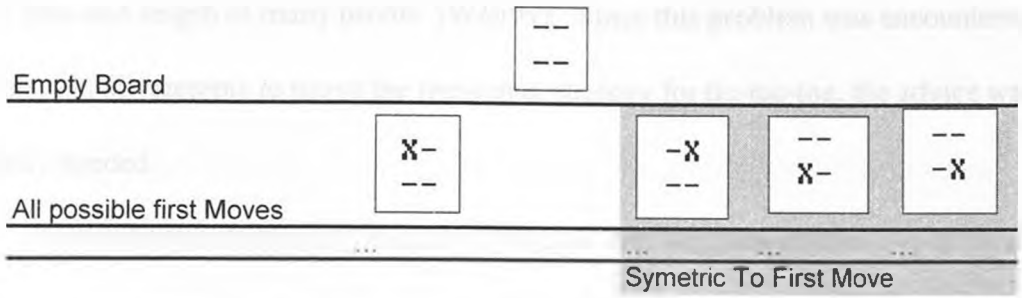

Figure 9. Illustration of tree-path elimination

Implementation Notes

(Note: Please see appendix D for the full implementation of ti-ta-to)

A great deal of the general structure for the ti-ta-to theory file is directly derived from the implementation of tic-tac-toe. However, several key changes were made in an effort to generate more workable proof situations. Specifically, anything that might require induction where it didn't really make any sense was eliminated. This includes things like the board, which was a dynamic list in tic-tac-toe and is fixed as a four-element vector of boardpositions in ti-ta-to. Moves, which were a pair of natural numbers in tic-tac-toe, are each a separate constructor for a "move" datatype. The player and boardposition datatypes remained unchanged.

In addition to firming up the representation of the game primitives, a good deal of energy was invested in simplifying and shortening operations on those primitives. All recursive functions that could possibly be implemented as non-recursive functions were re-written as such. Additionally, all "let" statements within the bodies of every function were eliminated if they could be. A message posted to the Isabelle-users mailing list stated that such statements, when used willy-nilly, often create an exponential explosion

in the state and length of many proofs [Wen99]. Since this problem was encountered first-hand in all attempts to prove the mini-max strategy for tic-tac-toe, the advice was carefully heeded.

One notable extension to the game of tic-tac-toe, implemented in ti-ta-to, was a function to determine whether or not a board was in a valid state. Specifically, it checks to make sure that, if one player had won the game, that the other player hadn't also "won" the game. Normally, this wouldn't be an issue in regular game play. However, proofs that instantiate all possible game boards by instantiating all possible combinations of boardpositions must necessarily generate boards that wouldn't occur during normal game play. So, the function that checks the state of a board must begin the each state analysis by ensuring that the board is not malformed.

Results

Like those for two-finger morra, the proofs for the symmetry of ti-ta-to were ver favorable. To begin with, a number of small proofs were completed to probe the validit of many of the key functions needed for the symmetry proof. Previous experiences writing tic-tac-toe and two-finger morra demonstrated that, no matter how correct the code appeared to be, it was almost never completely right from the onset. Small errors in programming or the omission of special case handling were the source of many hours of frustration. In this instance, however, the many verification proofs performed before embarking on the symmetry proof detected those small errors more quickly.

After the code describing ti-ta-to had been verified, the symmetry proof was written. Initially, a top-down approach to writing the proof was attempted. However, no

matter what angle it was attacked from, it always wound up in the same series of seemingly unresolvable sub-goals. "Seemingly" is used to describe them because, eventually, each was indeed solved. At that point in the proof, though, the lemmas required to resolve them hadn't been developed. Additionally, a bug in Isabelle itself prevented the development of some very straightforward proofs that could have been used to easily solve the problem.

Each of these seemingly unresolvable sub-goals was an absurdity that seemed quite easy to resolve. They included statements like "player X and player O can't both win the same game of ti-ta-to" and "the ti-ta-to board can not be both malformed and not malformed." Anyone who has ever played a game of tic-tac-toe knows that both of this statements are unequivocally true. However, Isabelle was not resolving them. If the proof were to be completed, it would be necessary to prove that these sorts of statements are actually true.

So, putting the top-down approach aside (it would be revisited later), the symmetries of individual functions used by the state-determining function were explored. In particular, this included the function used to determine whether or not a player has won the game, the function used to determine whether or not a board was full, and the function used to determine if the board was malformed or not. Each was proved with relative ease by doing rewriting and simplification. Once each of those proofs had been formalized, several of the "unresolvable" absurdities noted in the top-down proof were proven as standalone lemmas. With those lemmas in hand, the top-down proof was begun again. Only this time, once the absurdity resolution state of the proof was reached, each of the standalone absurdity lemmas were successfully applied to the resolution of previousl

"unresolvable" subgoals. With those out of the way, the proof of board symmetry was completed.

CONCLUSION

Summary of Results

While not the storming success that it could have been, using Isabelle to prove properties of games was a fairly successful venture. The mini-max strategy for tic-tac-toe, although it went nowhere, proved to be valuable learning experience. If nothing else, the lessons learned in the attempt went a long way towards easing the pain of formalizing and proving properties about the other two games investigated. For the game of two-finger morra, a successful proof of a winning strategy, one that works against opponents who play completely randomly, was developed and implemented. For ti-ta-to, the game board was shown to be symmetrical, a property useful in the development of fast tree searching algorithms.

Potential Avenues for Future Research

Naturally, the proof of the mini-max strategy is an obvious choice for future research. In light of the successes for two-finger morra and ti-ta-to, a solution seems entirely feasible. More than likely, the solution would incorporate the fixed-sized board representation used in ti-ta-to, as well as the explicit game play used in the winning strategy proof for two-finger morra. Both modifications to the structure of the game and proof representation would probably go a long way towards resolving issues related to the size of the statement being proven, as well as the problems with having to perform

induction on the size of the board. Given another two or three weeks, these modification could probably have been implemented and utilized. However, there is only so much time in a year. So, they'll have to wait for another day.

Additionally, the research done during this project could very well be applied to a multitude of different games. As has already been stated, the games of tic-tac-toe and ti-ta-to closely resemble many other familiar games like chess and othello. With a few modifications to the program source, results comparable to those already achieved could probably be had with a lesser degree of initial effort. Moreover, situations outside the realm of real games, like those studied by Von Neuman and Morganstern in the field of economics, could also be investigated and formalized within Isabelle. After all, they are based on the same principles that two-finger morra operates under.

Regardless of where this particular research project leads, it is encouraging to know that Isabelle and other generic theorem proving packages are still very much in development. Furthermore, as evidenced by the Isabelle users mailing list, there exists an army of researchers pushing the boundaries of what has been done within the confines of a generic logical framework. Even if this specific work were to remain insignificant, it is a comfort to know that there is an incredibly bright body of scientists continuall improving on the current state of automated formal verification.

APPENDIX A

TicTacToe.thy

```
(*************************************************************
 * Author : Michael (Beau) Burke
 * Date   : 5/24/2000
 * File   : TicTacToe.thy
 * Purpose: Formalization of the game Tic-Tac-toe
 *************************************************************)

TicTacToe = Main + Arith +

(* Tic-Tac-Toe primitive *)

datatype    player        = Xs | Os
datatype    boardposition = occupied player | empty
datatype    state         = won | draw | undefined | lost

(* more primitives and some wrappers for recdef'ed functions *)

types position            = "nat * nat"
      board               = boardposition list
      boardPlayerPair     = "board * player"
      natBoardPlayerTuple = "nat * boardPlayerPair"
      boardStatePair      = "board * state"
      move                = "player * position"

consts

      (* who'd have thought I'd have to _define_ nine! *)

      NINE :: nat

      (* some usefull boards *)

      emptyBoard      :: board
      occupiedBoard00 :: board
      occupiedBoard01 :: board
      occupiedBoard02 :: board
      occupiedBoard10 :: board
      occupiedBoard11 :: board
      occupiedBoard12 :: board
      occupiedBoard20 :: board
      occupiedBoard21 :: board
      occupiedBoard22 :: board
      xsWonBoard      :: board
      halfFullBoard   :: board

      (* list of all winning combinations *)

      winningPositions :: position list list
      isFull           :: board => bool

      (* functions on boards *)

      makeMove         :: board => move => board
      hasWon           :: board => player => bool
      getOpponent      :: player => player
      getState         :: board => player => state
```

```
      isBetterOrEqual  :: state => state => bool
      makeBestMove     :: boardPlayerPair => board
      makeBestMoveBody :: natBoardPlayerTuple => board
      exists           :: ('a => bool) => ('a list => bool)
      getEmpties       :: board => nat list
      getEmptiesBody   :: nat => board => nat list
      chooseBest       :: boardStatePair list => boardStatePair
      isWellFormed     :: board => bool
      count            :: 'a list => 'a => nat
      abs              :: nat => nat

      (* play to completion functions
         plays using function makeBestMove for both players *)

      playBestVSBest     :: boardPlayerPair => board
      playBestVSBestBody :: natBoardPlayerTuple => board

      playDumbVSDumb     :: boardPlayerPair => board
      playDumbVSDumbBody :: natBoardPlayerTuple => board

defs

NINE_def
"NINE == Suc (Suc (Suc (Suc (Suc (Suc (Suc ( 1 ))))))))"

emptyBoard_def
"emptyBoard ==
[
empty, empty, empty,
empty, empty, empty,
empty, empty, empty
]"


occupiedBoard00_def
"occupiedBoard00 ==
[
occupied(Os), empty, empty,
empty, empty, empty,
empty, empty, empty
]"


occupiedBoard01_def
"occupiedBoard01 ==
[
empty, occupied(Os), empty,
empty, empty, empty,
empty, empty, empty
]"


occupiedBoard02_def
"occupiedBoard02 ==
[
empty, empty, occupied(Os),
empty, empty, empty,
empty, empty, empty
]"
```

```
occupiedBoard10_def
"occupiedBoard10 ==
[
empty, empty, empty,
occupied(Os), empty, empty,
empty, empty, empty
]"


occupiedBoard11_def
"occupiedBoard11 ==
[
empty, empty, empty,
empty, occupied(Os), empty,
empty, empty, empty
]"


occupiedBoard12_def
"occupiedBoard12 ==
[
empty, empty, empty,
empty, empty, occupied(Os),
empty, empty, empty
]"


occupiedBoard20_def
"occupiedBoard20 ==
[
empty, empty, empty,
empty, empty, empty,
occupied(Os), empty, empty
]"


occupiedBoard21_def
"occupiedBoard21 ==
[
empty, empty, empty,
empty, empty, empty,
empty, occupied(Os), empty
]"


occupiedBoard22_def
"occupiedBoard22 ==
[
empty, empty, empty,
empty, empty, empty,
empty, empty, occupied(Os)
]"

xsWonBoard_def
"xsWonBoard ==
[
occupied(Xs),occupied(Os),occupied(Xs),
occupied(Xs),occupied(Xs),occupied(Os),
occupied(Xs),occupied(Os),occupied(Os)
]"
```

```
halfFullBoard_def
"halfFullBoard ==
[
empty, empty, empty,
occupied(Xs), occupied(Xs), empty,
occupied(Os), occupied(Os), occupied(Xs)
]"

(* all possible winning combinations for the game of
   Tic-Tac-Toe *)

winningPositions_def
"winningPositions ==
[
[(0,0), (0,1), (0,2)], [(1,0),(1,1),(1,2)], [(2,0),(2,1),(2,2)],
[(0,0), (1,0), (2,0)], [(0,1),(1,1),(2,1)], [(0,2),(1,2),(2,2)],
[(0,0), (1,1), (2,2)], [(2,0),(1,1),(0,2)]
]"

(* determines whether or not a board is full or
   not by checking each entry in the board.  If it
   finds a single empty space, it returns immediately
   with False *)

isFull_def
"isFull b ==
 let boardIsFull = False
 in let pred = (%e. if e = empty
                then boardIsFull = False
                else boardIsFull = True)
 in let results = List.list_all pred b
 in boardIsFull"

(* Inserts a boardposition into the space denoted by
   "mov," discarding whatever was there. *)

makeMove_def
"makeMove b mov ==
let (p,(x,y)) = mov in
( b[ x+x+x+y := occupied p ] )"

(* returns the opponent of the player argument *)

getOpponent_def
"getOpponent current == if current = Xs
                  then Os
                  else Xs"

(* determines if the player argument has won for the
   board argument.  It does so by checking each winning
   combination in winningPositioins against the actual
   board b.  If it finds a match, it returns with True *)

hasWon_def
"hasWon b p ==
let isP = (%h k. let (i,j) = k in case b!(i+i+i+j)
                of occupied someone  => if someone = h
                                        then True
                                        else False
                 | empty => False) in
let isPFun = %i. isP p i in
```

```
let checkOne = %positions. List.list_all isPFun positions in
exists checkOne winningPositions"

(* returns the state of the board argument for the player
   argument.  *)

getState_def
"getState b p ==
 let opponent = getOpponent p in
 let board_is_full = isFull b in
 let pstate = hasWon b p in
 let ostate = hasWon b opponent in
 if pstate
 then won
 else if ostate
         then lost
       else if board_is_full
              then draw
            else undefined"

(* comparitor function for board states.  Used to
   evaluate the desireability of one move over another *)

isBetterOrEqual_def
"isBetterOrEqual first second
 == case first of won                                    => True
                | draw      => (case second of won       => False
                                             | draw      => True
                                             | undefined => True
                                             | lost      => True)
                | undefined => (case second of won       => False
                                             | draw      => False
                                             | undefined => True
                                             | lost      => True)
                | lost      => (case second of won       => False
                                             | draw      => False
                                             | undefined => False
                                             | lost      => True)"


(* function to determine whether or not a board is
   well formed, i.e that the sum total of each player's
   moves is no more than one greater or less than their
   opponent *)

isWellFormed_def
"isWellFormed b == let xtotal = (count b (occupied(Xs))) in
                   let ototal = (count b (occupied(Os))) in
                   let difference = (abs (xtotal - ototal)) in
                   if (difference <= 1)
                   then True
                   else False"

(* absolute value function *)

abs_def
"abs n == if n<0 then (0-n) else n"

(* wrapper for getEmptiesBody *)

getEmpties_def
```

```
"getEmpties b == getEmptiesBody 0 b"

(* wrapper for playBestVSBestBody *)

playBestVSBest_def
"playBestVSBest pr ==
 let n = NINE
 in playBestVSBestBody (n, pr)"

(* wrapper for makeBestMoveBody *)

makeBestMove_def
"makeBestMove pr ==
 let n = NINE
 in makeBestMoveBody (n, pr)"

(* wrapper for playdumbVSDumbBody *)

playDumbVSDumb_def
"playDumbVSDumb pr ==
 let n = NINE
 in playDumbVSDumbBody (n, pr)"

(* returns a list of all empty positions in the board
   argument *)

primrec
getEmptiesBody_Nil "getEmptiesBody n [] = []"
getEmptiesBody_Cons "getEmptiesBody n (x#xs) = (let xsmties =
getEmptiesBody (Suc n) xs in
                                    if (x = empty)
                                    then (n#xsmties)
                                    else xsmties)"

(* counts the number of boardpositions which are occupied
   by the player argument *)

primrec
count_Nil "count [] p = 0"
count_Cons "count (x#xs) p = (let tailtotal = (count xs p) in
                              if x = p
                              then (Suc tailtotal)
                              else tailtotal)"

(* takes a list of board,state pairs and returns the board,state
   pair with the most desireable state *)

recdef chooseBest "measure ( %thelist. (length thelist))"
simpset "simpset() addsimps [isBetterOrEqual_def]"
"chooseBest (x#[]) = (x)"
"chooseBest (x1#x2#xs) = (let y = (chooseBest (x2#xs)) in
                      if isBetterOrEqual (snd x1) (snd y)
                          then x1
                          else y)"

primrec
exists_Nil "exists f [] = False"
exists_Cons "exists f (x#xs) = ( f(x) | exists f xs )"


(* mini-max implementation.  Takes a board and a player and
```

```
   makes the most advantageous move that can be made for
   that player *)

recdef makeBestMoveBody "measure ( %(n, (b, p)). ( n ) )"
"makeBestMoveBody (0, (b, p)) = b"
"makeBestMoveBody (n, (b, p)) =
 (let empties = (getEmpties b) in
  case empties of []     => b
                | (x#xs) => (let evaluateSingle =  ( %emptyspace.
                                 let newboard =
                                  b[emptyspace := occupied p] in
                                 let responseboard =
                                  makeBestMoveBody ((n-1),
                                  (newboard, (getOpponent p))) in
                                 let responsestate =
                                  (getState responseboard p) in
                                 (newboard, responsestate) ) in
                               let allresponses = List.map evaluateSingle
empties in
                               let bestpair = chooseBest allresponses in
                               (fst bestpair)) )"

(* pits the mini-max strategy against itself and plays a game of
   Tic-Tac-Toe to completion *)

recdef playBestVSBestBody "measure ( %(n, (b, p)). ( n ) )"
"playBestVSBestBody (0, (brd, plyr)) = brd"
"playBestVSBestBody (n, (brd, plyr)) =
 ( if isFull brd
   then brd
   else let newboard = makeBestMove (brd, plyr) in
        playBestVSBestBody ((n-1), (newboard, (getOpponent plyr)) ) )"

(* plays a game of Tic-Tac-Toe to completion by having each
   player choose the first move that that player can make *)

recdef playDumbVSDumbBody "measure ( %(n, (b, p)). ( n ) )"
"playDumbVSDumbBody (0, (brd, plyr)) = brd"
"playDumbVSDumbBody (n, (brd, plyr)) =
 ( let open = (getEmpties brd)
   in case open of [] => brd
                 | (x#xs) => playDumbVSDumbBody
                             ( (n-1),
                               (brd[x := occupied plyr],
                               (getOpponent plyr)) ) )"

end
```

TicTacToe.ML

```
(*************************************************************************
 * Author : Michael (Beau) Burke
 * Date   : 5/24/2000
 * File   : TicTacToe.ML
 * Purpose: Formalization of the game Tic-Tac-toe
 *************************************************************************)

(* some simple verification of the isFull function *)

Addsimps [isFull_def, emptyBoard_def, Let_def];
goal TicTacToe.thy "isFull emptyBoard = False";
by( Auto_tac );
qed "EmptyNotFull";
Delsimps [isFull_def, emptyBoard_def, Let_def];


(* verify that one can always lose a game of Tic-Tac-Toe, if
   one so chooses *)


Addsimps [isFull_def, emptyBoard_def, Let_def, getState_def,
isBetterOrEqual_def];
goal TicTacToe.thy
"isBetterOrEqual (getState (playBestVSBest (emptyBoard, Xs)) Xs) lost";
by( Auto_tac );
qed "OneCanAlwaysLoseOnEmpty";
Delsimps [getState_def, isBetterOrEqual_def, isFull_def, emptyBoard_def,
Let_def];


(* incomplete proof that playBestVSBest actually fills up a
   board


goal TicTacToe.thy "isFull (playBestVSBest (emptyBoard, Xs)) = True";
Addsimps [playBestVSBest_def, isFull_def, emptyBoard_def, Let_def];
by(Auto_tac);
by(simp_tac (simpset() addsimps [NINE_def]) 1);
by(simp_tac (simpset() addsimps playBestVSBestBody.rules) 1);
by(simp_tac (simpset() addsimps [getOpponent_def]) 1);
by( simp_tac (simpset() addsimps [makeBestMove_def]) 1);
by(simp_tac (simpset() addsimps [NINE_def]) 1);
by(simp_tac (simpset() addsimps makeBestMoveBody.rules) 1);

*)



(* incomplete proof that the mini-max strategy actually works

goal TicTacToe.thy
"isBetterOrEqual (getState (playBestVSBest (emptyBoard, Xs)) Xs) draw";
by (simp_tac (simpset() addsimps [playBestVSBest_def, Let_def]) 1);
by (simp_tac (simpset() addsimps [NINE_def]) 1);
by (simp_tac (simpset() addsimps [makeBestMove_def]) 1 );
by( simp_tac (simpset() addsimps [getOpponent_def]) 1 );
by( simp_tac (simpset() addsimps [emptyBoard_def]) 1);
by( simp_tac (simpset() addsimps [getState_def]) 1 );
```

```
by( simp_tac (simpset() addsimps [isFull_def]) 1);
by(simp_tac (simpset() addsimps [isBetterOrEqual_def]) 1);
by(simp_tac (simpset() addsimps [getOpponent_def]) 1);
by(Auto_tac);
Addsimps [hasWon_def];
by(Auto_tac);
Addsimps playBestVSBestBody.rules;
by(Auto_tac);
Addsimps [isFull_def];
by(Auto_tac);
Addsimps [getOpponent_def];
by(Auto_tac);
Addsimps [makeBestMove_def];
by(Auto_tac);
by(asm_full_simp_tac (simpset() addsimps [Let_def]) 1);
by (asm_full_simp_tac (simpset() addsimps [NINE_def]) 1);

*)
```

APPENDIX B

LEQ.thy

```
(*************************************************************
 * Author : Michael (Beau) Burke
 * Date   : 5/24/2000
 * File   : LEQ.thy
 * Purpose: Formalization of some properties about less-than-or
 *          equal-to
 *************************************************************)

LEQ = Nat + Arith +

(* surprisingly, Isabelle doesn't have very many proofs for the
   less-than-or-equal to operator.  Consequently, they are defined
   here.  The majority of the proofs in these files are merely "mirrors"
   of proofs done for the less-than operator.

end
```

LEQ.ML

```
(*************************************************************************
 * Author : Michael (Beau) Burke
 * Date   : 5/24/2000
 * File   : LEQ.ML
 * Purpose: Formalization of some properties about less-than-or
 *          equal-to
 *************************************************************************)

(* less than or equal to identity proof *)

Goal "(g1::nat) <= g1";
by(Auto_tac);
qed "LEQ_ident";

(* any number g1, minus a natural number, is less than or equal to
   the original number g1 *)

Goal "g1 - (n::nat) <= g1";
by(induct_tac "n" 1);
by(Auto_tac);
qed "Sub_Maint_LEQ";

(* any number g1, with an addition of any natural number, is less
   than or equal to the sum of g1 and that natural number *)

Goal "g1 <= g1 + (m::nat)";
by(induct_tac "m" 1);
by(Auto_tac);
qed "Add_Maint_LEQ";

val prem =
goal LEQ.thy "g - (Suc n) <= g - n";
by(simp_tac (simpset() addsimps prem) 1);
qed "Suc_N_Sub_LEQ";

Goal "[| a <= b; b <= c |] ==> a <= (c::nat)";
by(Asm_full_simp_tac 1);
qed "LEQ_trans";

val prem =
goal LEQ.thy "g1 <= g2 ==> g1 - (n::nat) <= g2";
by( resolve_tac [LEQ_trans] 1 );
by( resolve_tac prem 2 );
by( resolve_tac [Sub_Maint_LEQ] 1 );
qed "Sub_Maint_LEQ_strong";

val prem =
goal LEQ.thy "g1 <= g2 ==> g1 <= g2 + (n::nat)";
by( resolve_tac [LEQ_trans] 1 );
by( resolve_tac prem 1 );
by( resolve_tac [Add_Maint_LEQ] 1 );
qed "Add_Maint_LEQ_strong";

val prem =
goal LEQ.thy "g1 <= g2  ==> g1 - (n::nat) <= g2 + m";
by( resolve_tac [LEQ_trans] 1);
by( resolve_tac [Add_Maint_LEQ_strong] 2 );
by( resolve_tac prem 2 );
```

```
by( resolve_tac [Sub_Maint_LEQ_strong] 1);
by( resolve_tac [LEQ_ident] 1 );
qed "Arith_maint_LEQ";
```

APPENDIX C

TFM.thy

```
(*************************************************************
 * Author : Michael (Beau) Burke
 * Date   : 5/24/2000
 * File   : TFM.thy
 * Purpose: Formalization of the game Two-Finger Morra
 *************************************************************)

TFM = Main +  LEQ +

datatype fingers = ZeroF | OneF | TwoF
datatype call    = ZeroC | OneC | TwoC | ThreeC | FourC

types move = "fingers * call"
      game = "nat * nat"

consts  THREE           :: nat
        FIVE            :: nat
        FIFTEEN         :: nat
        winningPlay     :: move
        arbitraryPlay   :: fingers => call => move
        playRound       :: game => move => move  => game
        playNRounds     :: nat => game => game
        natValueF       :: fingers => nat
        natValueC       :: call => nat
        typeValueF      :: nat => fingers
        typeValueC      :: nat => call


defs

THREE_def
"THREE ==
 Suc (Suc (Suc 0))"

FIVE_def
"FIVE ==
 Suc (Suc (Suc (Suc (Suc 0))))"

FIFTEEN_def
"FIFTEEN ==
 Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (0))))))))))))))"

winningPlay_def
"winningPlay ==
 ( OneF, ThreeC )"

arbitraryPlay_def
"arbitraryPlay f c ==
 (f, c)"

playRound_def
"playRound g m1 m2 ==
 let (f1, c1) = m1      in
 let (f2, c2) = m2      in
 let (s1, s2) = g       in
```

```
 let ftotal   = (natValueF f1) + (natValueF f2) in
 if   ftotal = (natValueC c1)
 then if   ftotal = (natValueC c2)
      then g
      else (s1 + (natValueC c1), s2 - (natValueC c1))
 else if   ftotal = (natValueC c2)
      then (s1 - (natValueC c2), s2 + (natValueC c2))
      else g"


natValueF_def
"natValueF n ==
 case n of ZeroF => 0
         | OneF  => Suc 0
         | TwoF  => Suc (Suc 0)"

natValueC_def
"natValueC n ==
 case n of ZeroC  => 0
         | OneC   => Suc 0
         | TwoC   => Suc (Suc 0)
         | ThreeC => Suc (Suc (Suc 0))
         | FourC  => Suc (Suc (Suc (Suc 0)))"

typeValueF_def
"typeValueF f ==
 if f = 0
 then ZeroF
 else if f = (Suc 0)
      then OneF
      else TwoF"

typeValueC_def
"typeValueC c ==
 if c = 0
 then ZeroC
 else if c = (Suc 0)
      then OneC
      else if c = (Suc (Suc 0))
           then TwoC
           else if c = (Suc (Suc (Suc 0)))
                then ThreeC
                else FourC"


primrec
playNRounds_Nil "playNRounds 0 g = g"
playNRounds_Cons "playNRounds (Suc n) g =
 (let f      = (typeValueF ((Suc n) mod THREE)) in
  let c      = (typeValueC ((Suc n) mod FIVE)) in
  if f = OneF
  then g
  else if c = ThreeC
       then g
       else if f = ZeroF
            then g
            else if c = ZeroC
                 then g
                 else let (gp1,gp2) =
                        (playRound (0,0)
                          winningPlay (arbitraryPlay f c)) in
```

```
                    let (gp3,gp4) = (playNRounds n g) in
                    (gp1+gp3, gp2+gp4))"

end
```

TFM.ML

```
(*************************************************************
 * Author : Michael (Beau) Burke
 * Date   : 5/24/2000
 * File   : TFM.ML
 * Purpose: Formalization of the game Two-Finger Morra
 *************************************************************)

Addsimps
[FIVE_def,THREE_def,winningPlay_def,
 arbitraryPlay_def,typeValueF_def,typeValueC_def,
 natValueF_def,natValueC_def,Let_def,playRound_def];

val prems =
goal TFM.thy "g2 <= g1 ==> let (s1, s2) = \
\ playRound (g1,g2) winningPlay (arbitraryPlay F ZeroC) in \
\ s2 <= s1";
by(exhaust_tac "F" 1);
by(Auto_tac);
Addsimps prems;
by(Auto_tac);
by( resolve_tac [Sub_Maint_LEQ_strong] 1);
by( resolve_tac [LEQ_trans] 1);
by( resolve_tac prems 1 );
by( Auto_tac);
qed "Wins_on_ZeroC";
Addsimps [Wins_on_ZeroC];

val [first,second] =
goal TFM.thy "[| g2 <= g1; F ~= ZeroF |] ==> let (s1, s2) = \
\ playRound (g1,g2) winningPlay (arbitraryPlay F OneC) in \
\ s2 <= s1";
by(exhaust_tac "F" 1);
by(asm_full_simp_tac (simpset() addsimps [second]) 1);
by(Auto_tac);
by(resolve_tac [LEQ_trans] 1);
by(resolve_tac [Sub_Maint_LEQ_strong] 1);
by(resolve_tac [first] 1);
by(Auto_tac);
qed "Wins_on_OneC_except_ZeroF";
Addsimps [Wins_on_OneC_except_ZeroF];

val [first,second] =
goal TFM.thy "[| g2 <= g1; F ~= OneF |] ==> let (s1, s2) = \
\ playRound (g1,g2) winningPlay (arbitraryPlay F TwoC) \
\ in s2 <= s1";
by(exhaust_tac "F" 1);
by(asm_full_simp_tac (simpset() addsimps [second]) 2);
by(Auto_tac);
by(resolve_tac [LEQ_trans] 1);
by(resolve_tac [Sub_Maint_LEQ_strong] 1);
by(resolve_tac [first] 1);
by(Auto_tac);
qed "Wins_on_TwoC_except_OneF";
Addsimps [Wins_on_TwoC_except_OneF];

val prems =
goal TFM.thy "g2 <= g1 ==> let (s1, s2) = \
\ playRound (g1,g2) winningPlay (arbitraryPlay F ThreeC) in \
```

```
\ s2 <= s1";
by(Auto_tac);
qed "Wins_on_ThreeC";
Addsimps [Wins_on_ThreeC];

val prems =
goal TFM.thy "g2 <= g1 ==> let (s1, s2) = \
\ playRound (g1,g2) winningPlay (arbitraryPlay F FourC) in \
\ s2 <= s1";
by(exhaust_tac "F" 1);
by(Auto_tac);
by( resolve_tac [Sub_Maint_LEQ_strong] 1);
by( resolve_tac [LEQ_trans] 1);
by( resolve_tac prems 1 );
by( Auto_tac);
qed "Wins_on_FourC";
Addsimps [Wins_on_FourC];

val [first,second] =
goal TFM.thy "[| g2 <= g1; C ~= OneC |] ==> let (s1, s2) = \
\ playRound (g1,g2) winningPlay (arbitraryPlay ZeroF C) in \
\ s2 <= s1";
by( exhaust_tac "C" 1);
by(asm_full_simp_tac (simpset() addsimps [second]) 2);
by(Auto_tac);
qed "Wins_on_ZeroF_except_OneC";
Addsimps [Wins_on_ZeroF_except_OneC];

val [first,second] =
goal TFM.thy "[| g2 <= g1; C ~= TwoC |] ==> let (s1, s2) = \
\ playRound (g1,g2) winningPlay (arbitraryPlay OneF C) in \
\ s2 <= s1";
by(exhaust_tac "C" 1);
by(asm_full_simp_tac (simpset() addsimps [second]) 3);
by(Auto_tac);
qed "Wins_on_OneF_except_TwoC";
Addsimps [Wins_on_OneF_except_TwoC];

val prems =
goal TFM.thy "g2 <= g1 ==> let (s1, s2) = \
\ playRound (g1,g2) winningPlay (arbitraryPlay TwoF C) in \
\ s2 <= s1";
by( exhaust_tac "C" 1);
by(Auto_tac);
by(ALLGOALS (resolve_tac [LEQ_trans]));
by(resolve_tac [LEQ_ident] 8);
by(resolve_tac [LEQ_ident] 6);
by(resolve_tac [LEQ_ident] 4);
by(resolve_tac [LEQ_ident] 2);
by(ALLGOALS (resolve_tac [Sub_Maint_LEQ_strong]));
by(ALLGOALS (resolve_tac [LEQ_trans]));
by(Auto_tac);
qed "Wins_on_TwoF";
Addsimps [Wins_on_TwoF];


val prems  =
goal TFM.thy "\
\ let (a1,a2) = playRound (0,0) winningPlay (ZeroF, ZeroC) in \
\ let (b1,b2) = playRound (0,0) winningPlay (ZeroF, OneC) in \
\ let (c1,c2) = playRound (0,0) winningPlay (ZeroF, TwoC) in \
```

```
\ let (d1,d2) = playRound (0,0) winningPlay (ZeroF, ThreeC) in \
\ let (e1,e2) = playRound (0,0) winningPlay (ZeroF, FourC) in \
\ let (f1,f2) = playRound (0,0) winningPlay (OneF, ZeroC) in \
\ let (g1,g2) = playRound (0,0) winningPlay (OneF, OneC) in \
\ let (h1,h2) = playRound (0,0) winningPlay (OneF, TwoC) in \
\ let (i1,i2) = playRound (0,0) winningPlay (OneF, ThreeC) in \
\ let (j1,j2) = playRound (0,0) winningPlay (OneF, FourC) in \
\ let (k1,k2) = playRound (0,0) winningPlay (TwoF, ZeroC) in \
\ let (l1,l2) = playRound (0,0) winningPlay (TwoF, OneC) in \
\ let (m1,m2) = playRound (0,0) winningPlay (TwoF, TwoC) in \
\ let (n1,n2) = playRound (0,0) winningPlay (TwoF, ThreeC) in \
\ let (o1,o2) = playRound (0,0) winningPlay (TwoF, FourC) in \
\ (a2+b2+c2+d2+e2+f2+g2+h2+i2+j2+k2+l2+m2+n2+o2) \
\ < \
\ (a1+b1+c1+d1+e1+f1+g1+h1+i1+j1+k1+l1+m1+n1+o1)";
by(Auto_tac);
qed "Winning_has_advantage_when_equally_likely_at_empty";


val prems  =
goal TFM.thy "score2 < score1 ==>\
\ let (a1,a2) = playRound (score1,score2) winningPlay (ZeroF, ZeroC) in
\
\ let (b1,b2) = playRound (0,0) winningPlay (ZeroF, OneC) in \
\ let (c1,c2) = playRound (0,0) winningPlay (ZeroF, TwoC) in \
\ let (d1,d2) = playRound (0,0) winningPlay (ZeroF, ThreeC) in \
\ let (e1,e2) = playRound (0,0) winningPlay (ZeroF, FourC) in \
\ let (f1,f2) = playRound (0,0) winningPlay (OneF, ZeroC) in \
\ let (g1,g2) = playRound (0,0) winningPlay (OneF, OneC) in \
\ let (h1,h2) = playRound (0,0) winningPlay (OneF, TwoC) in \
\ let (i1,i2) = playRound (0,0) winningPlay (OneF, ThreeC) in \
\ let (j1,j2) = playRound (0,0) winningPlay (OneF, FourC) in \
\ let (k1,k2) = playRound (0,0) winningPlay (TwoF, ZeroC) in \
\ let (l1,l2) = playRound (0,0) winningPlay (TwoF, OneC) in \
\ let (m1,m2) = playRound (0,0) winningPlay (TwoF, TwoC) in \
\ let (n1,n2) = playRound (0,0) winningPlay (TwoF, ThreeC) in \
\ let (o1,o2) = playRound (0,0) winningPlay (TwoF, FourC) in \
\ (a2+b2+c2+d2+e2+f2+g2+h2+i2+j2+k2+l2+m2+n2+o2) \
\ < \
\ (a1+b1+c1+d1+e1+f1+g1+h1+i1+j1+k1+l1+m1+n1+o1)";
by(Auto_tac);
by(resolve_tac [less_trans] 1);
by(resolve_tac prems 1);
by(Auto_tac);
qed "Winning_has_advantage_when_equally_likely_at_winning";
```

APPENDIX D

TiTaToTyped.thy

```
(*************************************************************
 * Author : Michael (Beau) Burke
 * Date   : 5/24/2000
 * File   : TiTaToTyped.thy
 * Purpose: Formalization of a simplified Tic-Tac-Toe, dubbed
 *          "Ti-Ta-To"
 *************************************************************)

TiTaToTyped = Main +


datatype player        = Xs | Os
datatype boardposition = occupied player | empty
datatype state         = won | draw | undefined | lost | malformed
datatype position      = move00 | move01 | move10 | move11

types  board         = "boardposition * boardposition * boardposition *
boardposition"

consts      (* some usefull boards *)
      emptyBoard      :: board
      fullBoard       :: board
      occupiedBoard00 :: board
      occupiedBoard01 :: board
      occupiedBoard10 :: board
      occupiedBoard11 :: board
      xsWonBoard      :: board
        osWonBoard      :: board

      (* functions on boards *)
      isFull           :: board => bool
      makeMove         :: board => player => position => board
      hasWon           :: board => player => bool
      isMalformed      :: board => bool
      getOpponent      :: player => player
      getState         :: board => player => state
      isBetterOrEqual  :: state => state => bool
      rotate           :: board => board
      nrotate      :: nat => board => board

      (* helper functions *)
      count            :: board => player => nat
      difference       :: nat => nat => nat

      getEmpties   :: board => position list list
      getEmptiesFilter :: position list list => position list
        chooseBest       :: "(board * state) list => (board * state)"
      makeBestMove     :: "nat * ( board * player ) => board"

defs

emptyBoard_def
"emptyBoard ==
(
empty, empty,
empty, empty
```

```
)"

fullBoard_def
"fullBoard ==
(
occupied Xs, occupied Os,
occupied Xs, occupied Os
)"

occupiedBoard00_def
"occupiedBoard00 ==
(
occupied(Os), empty,
empty, empty
)"


occupiedBoard01_def
"occupiedBoard01 ==
(
empty, occupied(Os),
empty, empty
)"

occupiedBoard10_def
"occupiedBoard10 ==
(
empty, empty,
occupied(Os), empty
)"


occupiedBoard11_def
"occupiedBoard11 ==
(
empty, empty,
empty, occupied(Os)
)"

xsWonBoard_def
"xsWonBoard ==
(
occupied Xs, empty,
occupied Os, occupied Xs
)"

osWonBoard_def
"osWonBoard ==
(
occupied Os, empty,
occupied Xs, occupied Os
)"

isFull_def
"isFull b ==
 let (w,x,y,z) = b in
 if w = empty
 then False
 else if x = empty
      then False
      else if y = empty
```

```
            then False
            else if z = empty
                 then False
                 else True"


makeMove_def
"makeMove b ply pos ==
 let (w,x,y,z) = b in
 case pos of move00 => (occupied ply,x,y,z)
           | move01 => (w,occupied ply,y,z)
           | move10 => (w,x,occupied ply,z)
           | move11 => (w,x,y,occupied ply)"


getOpponent_def
"getOpponent current ==
 if current = Xs
 then Os
 else Xs"

hasWon_def
"hasWon b p ==
 if isMalformed b
 then False
 else
 let (w,x,y,z) = b in
 if (w = z)
      then if (w = (occupied p))
           then True
           else if x = y
                then if x = occupied p
                     then True
                     else False
                else False
      else if (x = y)
           then if (x = (occupied p))
                then True
                else False
           else False"


getEmpties_def
"getEmpties b ==
 let (w,x,y,z) = b in
 (if w = empty then [move00] else [])#(if x = empty then [move01] else
[])#
 (if y = empty then [move10] else [])#(if z = empty then [move11] else
[])#[]"

difference_def
"difference a b ==
 if a < b then b-a else a-b"

count_def
"count b p ==
 let (w,x,y,z) = b in
 (if w = occupied p then 1 else 0) +
 (if x = occupied p then 1 else 0) +
 (if y = occupied p then 1 else 0) +
```

```
 (if z = occupied p then 1 else 0)"


isMalformed_def
"isMalformed b ==
  if (let (w,x,y,z) = b in
     if (w = z)
     then if (w = (occupied Xs))
          then True
          else if x = y
               then if x = occupied Xs
                    then True
                    else False
               else False
     else if (x = y)
          then if (x = (occupied Xs))
               then True
               else False
          else False)
 then if (let (w,x,y,z) = b in
          if (w = z)
          then if (w = (occupied Os))
               then True
               else if x = y
                    then if x = occupied Os
                         then True
                         else False
                    else False
          else if (x = y)
               then if (x = (occupied Os))
                    then True
                    else False
               else False)
      then True
      else False
 else False"


getState_def
"getState b p ==
 if isMalformed b
 then malformed
 else if (hasWon b p)
      then won
      else if (hasWon b (getOpponent p))
           then lost
         else if (isFull b)
              then draw
             else undefined"

isBetterOrEqual_def
"isBetterOrEqual first second ==
 case first of won                                  => True
             | draw      => (case second of won      => False
                                          | draw     => True
                                          | undefined => True
                                          | lost     => True
                                          | malformed => True)
             | undefined => (case second of won      => False
                                          | draw     => False
                                          | undefined => True
```

```
                                          | lost      => True
                                          | malformed => True)
            | lost      => (case second of won       => False
                                          | draw      => False
                                          | undefined => False
                                          | lost      => True
                                          | malformed => True)
            | malformed => (case second of won       => False
                                          | draw      => False
                                          | undefined => False
                                          | lost      => False
                                          | malformed => True)"

rotate_def
"rotate b ==
 let (w,x,y,z) = b in
 (x,z,w,y)"


primrec
"nrotate 0 b = b"
"nrotate (Suc n) b = (rotate (nrotate n b))"


primrec
"getEmptiesFilter [] = []"
"getEmptiesFilter (x#xs) = (case x of [] => (getEmptiesFilter xs)
                                    | (y#ys) => (y#(getEmptiesFilter
xs)))"


recdef chooseBest "measure ( %thelist. length thelist )"
"chooseBest (x#xs) =
 (case xs of []     => x
          | (y#ys) => (if (isBetterOrEqual (snd x) (snd (chooseBest
xs)))
                      then x
                      else y))"


recdef makeBestMove "measure ( %(n, (b, p)). n )"
"makeBestMove (0, (b, ply)) = b"
"makeBestMove (Suc n, (b, ply)) =
 (let (w,x,y,z) = b in
 let empties = (getEmptiesFilter (getEmpties b)) in
 (case empties of [] => b
                | (a#as) => (fst
                             (chooseBest
                              (List.map
                               (%emptyspace.
                                (makeMove b ply emptyspace,
                                 (getState
                                  (makeBestMove
                                   (n,
                                    (makeMove b ply emptyspace,
getOpponent ply)
                                    )
                                   )
                                  ply)
                                 )
                                )
```

```
                                    empties)
                                   )
                                  )))"

end
```

TiTaToTyped.ML

```
(*************************************************************************
 * Author : Michael (Beau) Burke
 * Date   : 5/24/2000
 * File   : TiTaToTyped.thy
 * Purpose: Formalization of a simplified Tic-Tac-Toe, dubbed
 *          "Ti-Ta-To"
 *************************************************************************)


(*************************************************************
 * verification section.  These just check to make sure that
 * the code is doing sensible things
 *************************************************************)

Addsimps [isFull_def, emptyBoard_def, Let_def,
getState_def, isBetterOrEqual_def, hasWon_def,
emptyBoard_def, getOpponent_def,getEmpties_def,
xsWonBoard_def, osWonBoard_def, fullBoard_def,
isMalformed_def,difference_def,count_def];

goal TiTaToTyped.thy "isFull emptyBoard = False";
by( Auto_tac );
qed "EmptyNotFull";

goal TiTaToTyped.thy "isFull fullBoard = True";
by(Auto_tac);
qed "FullIsFull";

goal TiTaToTyped.thy "hasWon emptyBoard Xs = False";
by(Auto_tac);
qed "hasWonLosesCheck";

goal TiTaToTyped.thy "hasWon xsWonBoard Xs = True";
by(Auto_tac);
qed "hasWonWinsCheck";

goal TiTaToTyped.thy "hasWon fullBoard p = False";
by(exhaust_tac "p" 1);
by(Auto_tac);
qed "hasWonCheck";

goal TiTaToTyped.thy "getState fullBoard p = draw";
by(exhaust_tac "p" 1);
by(Auto_tac);
qed "getStateCheck";

goal TiTaToTyped.thy "getState xsWonBoard Os = lost";
by(Auto_tac);
qed "getStateCheck2";

Delsimps [isFull_def, emptyBoard_def, Let_def,
getState_def, isBetterOrEqual_def, hasWon_def,
emptyBoard_def, getOpponent_def,getEmpties_def,
xsWonBoard_def, osWonBoard_def, fullBoard_def,
isMalformed_def,difference_def,count_def];

Goalw [getOpponent_def] "p = getOpponent (getOpponent p)";
by(exhaust_tac "p" 1);
```

```
by(Auto_tac);
qed "OpponentReflect";


Goalw [getOpponent_def] "p ~= getOpponent p";
by(Auto_tac);
qed "OpponentDifferent";


goal TiTaToTyped.thy  "(hasWon (w,x,y,z) p) --> w = z | x = y";
by(asm_full_simp_tac (simpset() addsimps [hasWon_def,Let_def])1);
qed "hasWonFOL";

Goal "[| w = empty | x = empty | y = empty | z = empty |]\
\      ==> getState (w,x,y,z) p ~= draw";
by(asm_full_simp_tac (simpset() addsimps [getState_def])1);
by(asm_full_simp_tac (simpset() addsimps
[isFull_def,getOpponent_def])1);
by(asm_full_simp_tac (simpset() addsimps [Let_def])1);
by(asm_full_simp_tac (simpset() addsimps [difference_def,count_def])1);
by(Auto_tac);
qed "OneEmptyNotDraw";


Goal "(getState (w,x,y,z) p) = draw ==> isFull (w,x,y,z)";
by(ALLGOALS (exhaust_tac "w"));
by(ALLGOALS (exhaust_tac "x"));
by(ALLGOALS (exhaust_tac "y"));
by(ALLGOALS (exhaust_tac "z"));
by(ALLGOALS (exhaust_tac "p"));
by(ALLGOALS (asm_full_simp_tac (simpset() addsimps [isFull_def])));
by(ALLGOALS (asm_full_simp_tac (simpset() addsimps [OneEmptyNotDraw])));
by(ALLGOALS (asm_full_simp_tac (simpset() addsimps [getState_def])));
by(ALLGOALS
 (asm_full_simp_tac (simpset() addsimps [hasWon_def,getOpponent_def])));
by(ALLGOALS (asm_full_simp_tac (simpset() addsimps [Let_def])));
qed "DrawStateOnlyOnFullBoard";


(*************************************************************
 * end of verification section.
 *************************************************************)

(* Proof of board symmetry
 *************************************************************)

(* hasWon symmetries *)

goal TiTaToTyped.thy "hasWon (w,x,y,z) p --> hasWon (rotate (w,x,y,z))
p";
by(asm_full_simp_tac
 (simpset() addsimps [rotate_def,hasWon_def,isMalformed_def,Let_def])1);
qed "WinIsSymetric";

goal TiTaToTyped.thy "hasWon (rotate (w,x,y,z)) p --> hasWon (w,x,y,z)
p";
by(asm_full_simp_tac
 (simpset() addsimps [rotate_def,hasWon_def,isMalformed_def,Let_def])1);
qed "WinIsSymetricConverse";
```

```
goal TiTaToTyped.thy "~ hasWon (rotate (w,x,y,z)) p --> ~ hasWon
(w,x,y,z) p";
by(asm_full_simp_tac (simpset() addsimps
 [isMalformed_def,hasWon_def,rotate_def,Let_def])1);
qed "NotWinIsSymetric";

goal TiTaToTyped.thy "~ hasWon (w,x,y,z) p --> ~ hasWon (rotate
(w,x,y,z)) p";
by(asm_full_simp_tac (simpset() addsimps
 [isMalformed_def,hasWon_def,rotate_def,Let_def])1);
qed "NotWinIsSymetricConverse";


(* isFull symmetries *)

goal TiTaToTyped.thy "isFull (w,x,y,z) --> isFull (rotate (w,x,y,z))";
by(asm_full_simp_tac (simpset() addsimps
[isFull_def,rotate_def,Let_def])1);
qed "IsFullIsSymetric";

goal TiTaToTyped.thy "isFull (rotate (w,x,y,z)) --> isFull (w,x,y,z)";
by(asm_full_simp_tac (simpset() addsimps
[isFull_def,rotate_def,Let_def])1);
qed "IsFullIsSymetricConverse";

goal TiTaToTyped.thy "~ isFull (rotate (w,x,y,z)) --> ~ isFull
(w,x,y,z)";
by(asm_full_simp_tac (simpset() addsimps
[isFull_def,rotate_def,Let_def])1);
qed "NotIsFullIsSymetric";

goal TiTaToTyped.thy "~ isFull (w,x,y,z) --> ~ isFull (rotate
(w,x,y,z))";
by(asm_full_simp_tac (simpset() addsimps
[isFull_def,rotate_def,Let_def])1);
qed "NotIsFullIsSymetricConverse";


(* isMalformed Symmetries *)

goal TiTaToTyped.thy
"isMalformed (w,x,y,z) --> isMalformed (rotate (w,x,y,z))";
by(asm_full_simp_tac (simpset() addsimps
  [isMalformed_def,hasWon_def,rotate_def,Let_def])1);
qed "IsMalformedIsSymetric";

goal TiTaToTyped.thy
"isMalformed (rotate (w,x,y,z)) --> isMalformed (w,x,y,z)";
by(asm_full_simp_tac (simpset() addsimps
  [isMalformed_def,hasWon_def,rotate_def,Let_def])1);
qed "IsMalformedIsSymetricConverse";

goal TiTaToTyped.thy
"~ isMalformed (rotate (w,x,y,z)) --> ~ isMalformed (w,x,y,z)";
by(asm_full_simp_tac (simpset() addsimps
  [isMalformed_def,hasWon_def,rotate_def,Let_def])1);
qed "NotIsMalformedIsSymetric";

goal TiTaToTyped.thy
"~ isMalformed (w,x,y,z) --> ~ isMalformed (rotate (w,x,y,z))";
by(asm_full_simp_tac (simpset() addsimps
```

```
  [isMalformed_def,hasWon_def,rotate_def,Let_def])1);
qed "NotIsMalformedIsSymetricConverse";


(* uniquness of winning state *)

goal TiTaToTyped.thy
"hasWon (w,x,y,z) p --> ~ hasWon (w,x,y,z) (getOpponent p)";
by(asm_full_simp_tac (simpset() addsimps
  [hasWon_def,getOpponent_def,isMalformed_def,Let_def])1);
by(ALLGOALS (exhaust_tac "p"));
by(Auto_tac);
qed "OneWinsTheOtherLoses";


goal TiTaToTyped.thy
"hasWon (w,x,y,z) p --> ~ hasWon (rotate (w,x,y,z)) (getOpponent p)";
by(asm_full_simp_tac (simpset() addsimps
  [rotate_def,hasWon_def,getOpponent_def,isMalformed_def,Let_def])1);
by(ALLGOALS (exhaust_tac "p"));
by(Auto_tac);
qed "OneWinsTheOtherLosesEvenWhenRotated";


(* proofs specifying absurdities *)

Goal "[| hasWon (w,x,y,z) p; hasWon (w,x,y,z) (getOpponent p) |] ==>
False";
by(asm_full_simp_tac (simpset() addsimps [OneWinsTheOtherLoses])1);
qed "HasWonAbsurdity1";

Goal "[| ~ hasWon (w,x,y,z) p; hasWon (rotate (w,x,y,z)) p |] ==>
False";
by(asm_full_simp_tac (simpset() addsimps [NotWinIsSymetricConverse])1);
qed "HasWonAbsurdity2";

Goal "[| isMalformed (w,x,y,z); ~ isMalformed (rotate (w,x,y,z)) |] ==>
False";
by(asm_full_simp_tac (simpset() addsimps [IsMalformedIsSymetric])1);
qed "IsMalformedAbsurdity1";

Goal "[| ~ isMalformed (w,x,y,z); isMalformed (rotate (w,x,y,z)) |] ==>
False";
by(asm_full_simp_tac (simpset() addsimps
[NotIsMalformedIsSymetricConverse])1);
qed "IsMalformedAbsurdity2";

Goal "[| ~ isFull (w,x,y,z); isFull (rotate (w,x,y,z)) |] ==> False";
by(asm_full_simp_tac (simpset() addsimps
[NotIsFullIsSymetricConverse])1);
qed "IsFullAbsurdity1";


(* proof of symmetry for a single rotation *)

goal TiTaToTyped.thy
"getState (w,x,y,z) p = getState (rotate (w,x,y,z)) p";
by(simp_tac (simpset() addsimps

[getState_def,Let_def,IsMalformedIsSymetric,WinIsSymetric,IsFullIsSymetr
ic])1);
```

```
by(Auto_tac);
by(resolve_tac [HasWonAbsurdity1] 1);
by(Auto_tac);
by(resolve_tac [HasWonAbsurdity1] 7);
by(Auto_tac);
by(resolve_tac [IsFullAbsurdity1] 10);
by(Auto_tac);
by(resolve_tac [HasWonAbsurdity2] 10);
by(Auto_tac);
by(resolve_tac [HasWonAbsurdity2] 10);
by(Auto_tac);
by(resolve_tac [HasWonAbsurdity2] 1);
by(Auto_tac);
by(resolve_tac [HasWonAbsurdity2] 3);
by(Auto_tac);
by(resolve_tac [HasWonAbsurdity2] 5);
by(Auto_tac);
by(resolve_tac [IsMalformedAbsurdity2] 1);
by(Auto_tac);
by(resolve_tac [IsMalformedAbsurdity2] 1);
by(Auto_tac);
by(resolve_tac [IsMalformedAbsurdity2] 2);
by(Auto_tac);
by(resolve_tac [IsMalformedAbsurdity2] 2);
by(Auto_tac);
by(resolve_tac [IsMalformedAbsurdity2] 2);
by(Auto_tac);
by(resolve_tac [IsMalformedAbsurdity2] 2);
by(Auto_tac);
by(resolve_tac [HasWonAbsurdity2] 1);
by(Auto_tac);
qed "TiTaToBoardIsSymetric";
```

BIBLIOGRAPHY

[Bur63] Ewald Burger. *Introduction to the Theory of Games*. trans. John E. Freund. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1963.

[Bur97] Burke, Robin. *Introduction to Artificial Intelligence Representation and Memory (Lecture Notes* http://www.cs.uchicago.edu/~burke/cs250/lectures/lecture1013.html

[Lag96] Lagarias, Jeff. *The 3x+1 problem and its generalizations*. http://www.cecm.sfu.ca/organics/papers/lagarias/index.html CECM/IMpress (Simon Fraser University). 1996

[Nip99] *Nipkow, Tobias*. Tutorial on Isabelle/HOL. Http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle99/doc/tutorial.pdf October 31, 1999

[Pau94] Lawerence C. Paulson. *Isabelle A Generic Theorem Prover*. Springer-Verlag, Berlin, 1994.

[Pau99] Paulson, Lawerence. *The Isabelle Reference Manual*. http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle99/doc/ref.pdf October 31, 1999

[Pfe96] Frank Pfenning. The Practice of Logical Frameworks. in *Trees in Algebra and Programming - CAAP'96*. editor Helene Kirchner. Springer-Verlag, Berlin, 1996.

[S-AJ97] Rolf Socher-Ambrosious and Patricia Johann. *Deduction Systems*. Springer-Verlag, New York, New York, 1997.

[Set97] Ravi Sethi. *Programming Languages: Concepts & Constructs*. Addison-Wesley, 1997

[Sta99] Saul Stahl. *A Gentle Introduction to Game Theory*. The American Mathematical Society, Providence, Rhode Island, 1999.

[Wei98] Mark Allen Weiss. *Data Structures & Problem Solving Using Java*. Addison- Wesley, 1998.

[Wen99] Wenzel, Markus. *Re: Strange error message* (from Isabelle users mailing list archive). ftp://ftp.cl.cam.ac.uk/ml/isabelle-users.99.gz. September 20, 1999.