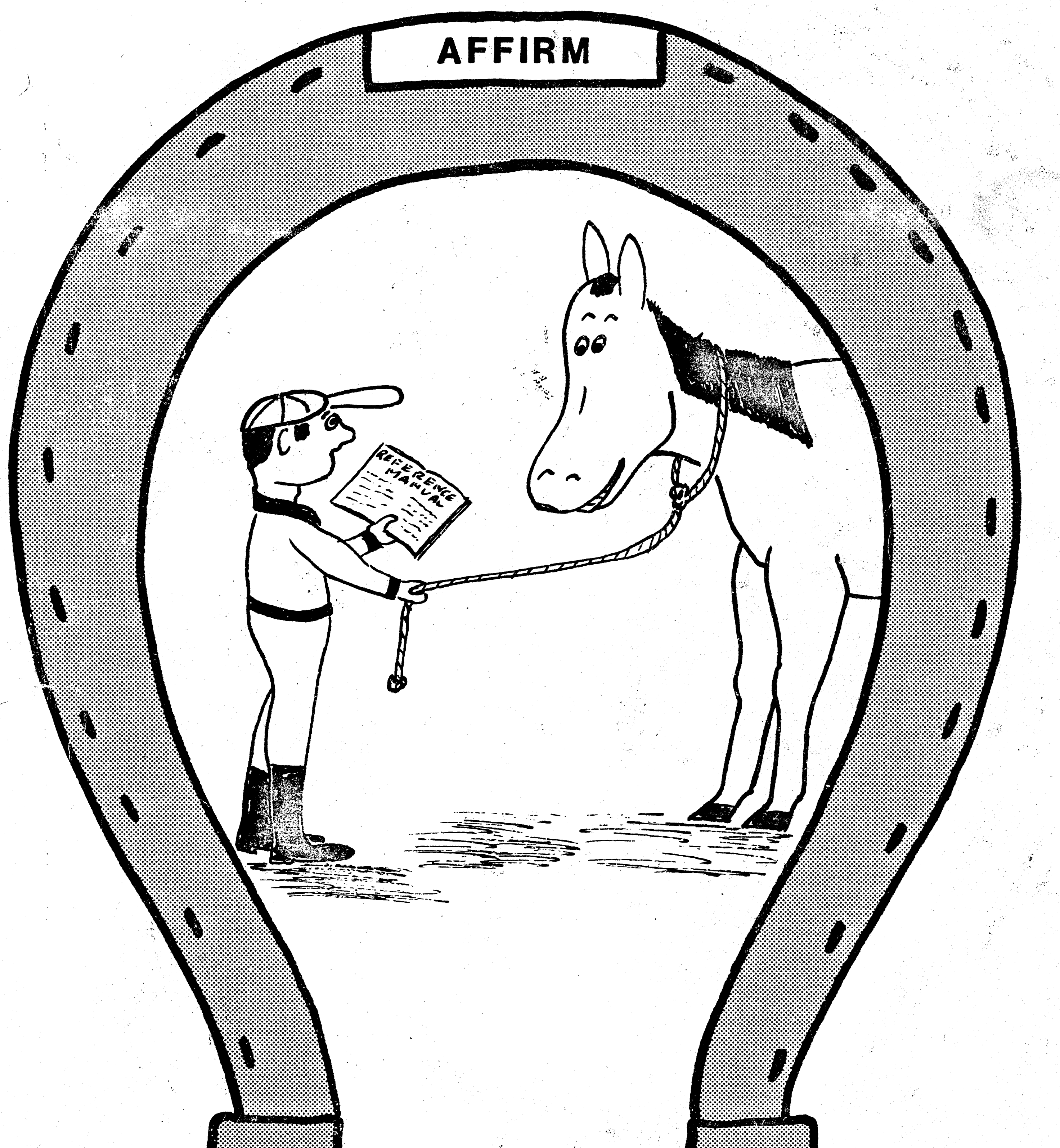


AFFIRM Reference Manual

David H. Thompson and Roddy W. Erickson, Editors

AFFIRM

Reference Manual

David H. Thompson and Roddy W. Erickson, Editors

Version 2.0 - February 19, 1981

Corresponds to *AFFIRM* Version 1.21

USC Information Sciences Institute

4676 Admiralty Way

Marina Del Rey, California 90291

(213) 822-1511 . ARPANET: AFFIRM@ISIF

The AFFIRM Reference Library

AFFIRM is an experimental interactive system for the specification and verification of abstract data types and programs. It was developed by the Program Verification Project at the USC Information Sciences Institute (ISI) for the Defense Advanced Research Projects Agency. The Reference Library is composed of five documents:

Reference Manual
: A detailed discussion of the major concepts behind ***AFFIRM*** presented in terms of the abstract machines forming the system's structure as seen by the user.

Users Guide
: A question-and-answer dialogue detailing the whys and wherefores of specifying and proving using ***AFFIRM***.

Type Library
: A listing of several abstract data types developed and used by the ISI Program Verification Project. The data type specifications are maintained in machine-readable form as an integral part of the system.

Annotated Transcripts
: A series of annotated transcripts displaying ***AFFIRM*** in action, to be used as a sort of workbook along with the Users Guide and Reference Manual.

Collected Papers
: A collection of articles authored by members of the ISI Program Verification Project (past and present), as well as an annotated bibliography of recent papers relevant to our work.

Program Verification Project Members

The USC/Information Sciences Institute Program Verification Project is headed by Susan L. Gerhart, with members Roddy W. Erickson, Stanley Lee, Lisa Moses, and David H. Thompson. Past project members include Raymond L. Bates, Ralph L. London, David R. Musser, David G. Taylor, and David S. Wile.

Cover designs by Nelson Lucas.

Special dedication to Affirmed, the only race horse named after a verification system.

This research was supported by the Defense Advanced Research Projects Agency and the Rome Air Defense Command. Views and conclusions are the authors'.

Abstract

Affirm is an experimental interactive system for the development of specifications and the verification of abstract data types and algorithms. This document discusses the major concepts behind ***Affirm***, and explains the purpose and use of each of the abstract machines comprising the structure of the system as seen by the user.

Acknowledgements

This manual was written by D. A. Baker[1], R. L. Bates, R. W. Erickson, S. L. Gerhart, M. L. Horowitz[2], S. Lee, R. L. London, L. Moses, D. R. Musser[3], D. G. Taylor, D. H. Thompson, and D. S. Wile.

In addition, J. V. Guttag and D. S. Lankford heavily influenced the design and development of ***Affirm***. Numerous colleagues provided valuable feedback from system demonstrations.

AffirmED Sent to Stud

[NEW YORK/October 22, 1979] Harbor View Farm's Affirmed, the leading money winner of all time in thoroughbred horse racing, has been retired effective immediately, trainer Laz Barrera announced today. The 4-year-old colt will be sent to Spendthrift Farm in Lexington, Ky., for stud duty.

Affirmed, who won the 1978 Triple Crown, recorded 22 victories--19 stakes--five seconds and one third in 29 career starts.[4]

Clarke's Law of Research: Every revolutionary idea ... evokes three stages of reaction in the listener:

- "It's completely impossible."
- "It's possible, but highly impractical."
- "I said it all along."

[1]Computer Science Department, University of Southern California, Los Angeles, CA 90007.

[2]Present address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

[3]Present address: Computer Science Branch, General Electric Research and Development Center, Schenectady, NY 12345.

[4]From the Los Angeles Times, October 22, 1979.

Suggestions about and criticisms of content, style, and organization of this manual should be addressed to one of the editors, Information Sciences Institute, 4676 Admiralty Way, Marina Del Rey, California, USA 90291, ARPANET address Affirm@ISIF, (213) 822-1511.

Table of Contents

1. Introduction

Affirm is an interactive system for the specification and verification of abstract data types and programs. It accepts algebraic specifications of data abstractions [Guttag 75, Guttag 78a] and programs written in a variant of Pascal [Jensen 75] extended with several features from ***Euclid*** [Lampson 77, London 78]. The system contains a verification condition generator which supports the standard inductive assertion method [Floyd 67] as well as the subgoal assertion method [Morris 77]. ***Affirm*** also contains a natural deduction theorem prover for interactive proof of verification conditions and properties of data abstractions. The system provides the rudimentary capabilities for organizing large specifications and collections of axiomatic and derived properties in an online data base for retrieval during subsequent program or data abstraction verification.

Affirm is implemented in Interlisp [Teitelman 78] and runs under the ***Tenex*** and ***Tops-20*** operating systems. It is the successor to (and combines features of) two previous systems, the ***Xivus*** System [Good 75] and ***Dtvs*** [Musser 77, Guttag 78b], the Data Type Verification System.

Affirm's theorem prover is based on the use of rewrite rules [Musser 77, Musser 80]. Given a statement to be proved, the rules "reduce" or "simplify" the expression as far as possible by replacing instances of left hand sides of axioms of data types by the corresponding right hand sides. This process requires that the axioms of each data type have an appropriate form in order to avoid loops in rewriting. They must produce the same results independent of order of application, and must cover enough cases of reduction. Various methods, both informal and implemented, are used to check and improve the rewriting behavior of a given set of axioms.

1.1. Proving: Human vs. Machine

A mechanical proof looks very much like any mathematical proof. The user must state the theorem, find and state lemmas, indicate how and when these lemmas enter the proof, establish appropriate subgoals, reduce the complexity of intermediate steps by throwing away irrelevant information, etc. The user may also state induction schemas, for example structural induction, by which the system sets up the steps of an induction proof. Recording proof steps, undoing disastrous steps, and redoing previous proof steps are all performed by the system. Functions may be expressed recursively and then the definitions invoked explicitly during a proof. In short, the user must find the right set of axioms, the theorems to be proved, and the lemma structure of a proof. A great amount of planning must go into such a proof, since the system makes no effort to find proofs for the user beyond applying the rules of the axioms (with the exception of algorithms for finding equality chains and instantiations of lemmas).

The earlier view of a mechanical program verification system was more heavily oriented toward programs and verification condition generation. In contrast, ***Affirm*** treats the verification condition generator as a more subordinate component because there are numerous properties besides verification conditions to prove. The system is evolving further to support organized bodies of knowledge about types, as well as a whole calculus of programs.

1.2. Organization of this Manual

Chapter 2 discusses the procedure for obtaining access to ***Affirm***, describes the system structure of ***Affirm*** as an interacting set of abstract machines, and provides a brief overview of each abstract machine. Each of the remaining chapters deals with the details of one of the abstract machines.

Appendix I contains a description of the syntax of user commands. Appendix II contains the grammar of the programming language currently processed by ***Affirm***: Pascal with extensions. Appendix III details some of ***Affirm***'s dependencies on Interlisp. Appendix IV contains examples of the verification conditions generated for most statement constructs in the accepted language. Appendix V contains a compendium of restrictions, outright bugs, and "curious features" (pronounced with a Transylvanian accent). Appendix VI contains brief summaries of some of the available commands in the operating system executives under which ***Affirm*** runs (***Tenex*** and ***Tops-20***). Appendix VII contains a glossary of terms. Appendix VIII contains a list of commands most useful for users new to the system. Appendix IX categorizes the commands according to function, within each abstract machine. And finally, Appendix X contains the synopses of ***Affirm*** commands.

1.3. Character Set Conventions Used in this Manual

1.3.1. Prose

Both <u>underlining</u> and *italics* are used for emphasis. Underlining is also used to distinguish ***Affirm*** command names (such as the <u>axiom</u> command) in running prose. **Bold face** is used to demarcate control characters.

1.3.2. Examples

Examples are offset from the running text. *Italics* are used to display nonterminal symbols. Both the normal font and a `typewriter-like font` are used to display language symbols and keywords. Square brackets [] denote optionality of the enclosed items, and ellipses ... signify a possibly empty list. As an example, the syntax of the <u>axiom</u> command is

`axiom[s]` *equation*$_1$[, ..., *equation*$_n$];

2. System Structure: An Overview

The ***Affirm*** system can be viewed as a collection of abstract machines interacting with each other in various ways. Figure 2-1 displays the overall structure of ***Affirm***. Each of the following sections briefly describes the workings of a particular abstract machine.

It should be emphasized that most ***Affirm*** commands affect more than one abstract machine, even though their main function places them in a particular machine. For example, the axiom command is a specification machine transformation, and is described in Chapter 4. But the same axiom command also has some effects on the Rewrite Rule machine, as is described in Chapter 5.

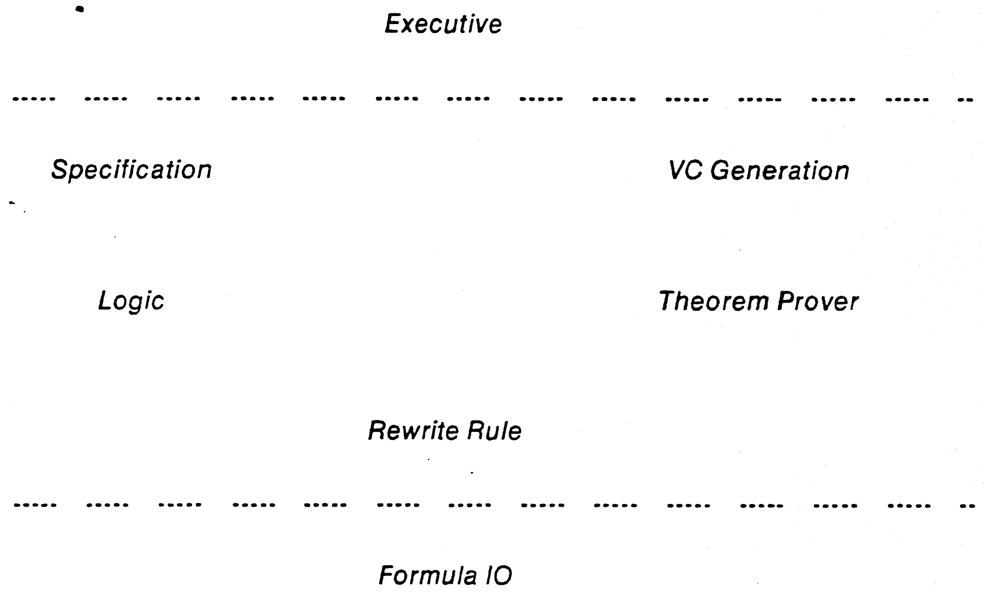


Figure 2-1: ***AFFIRM*** system structure

2.1. The Executive

Described in Chapter 3, the executive binds the various abstract machines into one huge system. It performs the basic command recognition and parameter input processes, and contains several small machines that provide services to the rest of ***Affirm***, as well as to the user. These small machines are the spelling corrector, the user profile mechanism, the help system, the monitor and timing system, the gripe facility, and the news facility. Each of these small machines is described in the chapter discussing the executive.

The executive processes errors, whether they arise from internal code errors, user interrupts, or user-supplied input. The executive also handles communication with the underlying operating systems and with various text editors.

2.2. The Specification Machine

Described in Chapter 4, this machine builds the abstract data type specifications and performs type checking on all input to the system. Commands are provided for creating, modifying, destroying, re-creating, saving, reading, and printing type specifications. Most of these capabilities are also provided for manipulating each of the objects comprising the structure within types.

2.3. The Rewrite Rule Machine

Described in Chapter 5, this machine rewrites expressions based on the set of currently defined rewrite rules. This powerful facility is used by many parts of the system for expression manipulation.

2.4. The Logic Machine

Described in Chapter 6, this machine provides the basic underlying propositional calculus operations, as well as skolemization, normalization, unification, instantiation, and case analysis.

2.5. The Theorem Prover

Described in Chapter 7, this machine maintains the *proof structure*, which is a forest (really a directed acyclic graph) of propositions. Many operations are provided for moving about within the tree associated with each theorem. There are also quite a few commands which perform transformations of the tree -- adding, removing, or modifying structure. It all adds up to a proof!

2.6. The VC Generation Machine

Described in Chapter 8, this machine oversees the generation of verification conditions from programs that have been read and type-checked.

2.7. The Formula I/O Machine

Described in Chapter 9, this machine oversees the printing of complex propositions and expressions and performs file I/O functions. The user has very little to say about the format of the printed output at present. Parenthesization protocol and operator priorities are not modifiable by the user. Such things as listing formats of the various components of a type specification are somewhat controllable via user profile entries [§3.13].

3. The Executive

3.1. Basic Command Processing

Interaction with ***Affirm*** is initiated through its command language. Each command is of the form

commandName commandParameters ;

where the form of the command parameters depends on the particular command name. Parameters are expressed in a language similar to that of most programming languages in its conventions for names, numeric constants, infix and prefix operators, operator priority, etc. Commands are used to direct the system to accept new data type specifications; to read Pascal programs from a file; to direct the theorem prover; and to perform general utility functions. All commands warn the user if any excess parameters are supplied. Comments can be interspersed anywhere in the command line, except inside other comments: nesting is not supported. Comments are enclosed in curly braces '{' and '}'; they cannot contain a right curly brace.

All commands may either be typed in directly or may be prepared on a file and then read in using the read command.

3.2. Affirm-User Interactions

We have spent a fair amount of effort attempting to make ***Affirm*** more or less *user-habitable* (whatever that means). To that end, input to the system is immediately correctable; the user can type ahead; the system performs spelling correction on certain objects (e.g. type names and command names); commands can be undone, fixed, and/or redone; etc. This section describes the various flavors of user interaction with ***Affirm***.

3.2.1. Edit Characters During Input

Several control characters are useful for editing a command and its parameters while typing.

Function	*Tops-20*	*Tenex*
delete character	DEL	control-A
delete line	control-U	control Q
re-display line	control-R	control-R
immediate input buffer delete	control-Z	DEL

3.2.2. Basic System Actions

Affirm allows and encourages typeahead. However, there are several instances where the system will query the user, where the user will not usually have anticipated that the interaction would occur. In such cases, ***Affirm*** saves any information typed ahead, and proceeds to carry out the "unexpected" interaction. It then restores the input buffer, and continues normally. During the interaction, **control-Z**[6] will *not* cause the saved input buffer to be flushed.

There are several instances where such an unexpected interaction may take place. The most notable is during the execution of the algorithm that determines rewrite rule convergence, when rules are generated by the algorithm in an attempt to restore unique termination in a set of rewrite rules [Chapter 5].

This manual notes instances of unexpected interactions in the description of each command that may cause the interaction.

3.2.3. Responding to System Questions

All of the interactive questions, whether unexpected or not, can be queried for a list of the responses by typing a question mark ?. In addition, many of the questions require only a single letter, or only enough letters of a particular response to make it distinct from all other choices. ***Affirm*** will then pause and wait for user confirmation. The user confirms a response by typing a blank or carriage return. The user can type DEL[7] at any time before confirmation of one response in order to reset the response protocol to the beginning. The user can then choose a different response.

3.3. Spelling Correction

Affirm keeps various lists of command names, type names, theorem names, etc., in order to attempt to correct misspelled identifiers. This facility is quite useful because it greatly enhances ***Affirm***'s flexibility in accepting input.

The Basic Interaction Protocol

The spelling corrector initially attempts to obtain a small set of possible respellings of the misspelled identifier, where each respelling looks like the misspelled identifier. The size of this set and the closeness of the lookalikes are heuristic parameters that may change at any time.[8]

[6]DEL under ***Tenex***.

[7]control-A under ***Tenex***.

[8]The respelling set currently has a maximum size of 3; a word looks like another word if the first matches the second, letter for letter, at least 40% of the time. The definition of "match letter for letter" is not exact, as simple letter transpositions count as matching. The exact algorithm is Teitelman's [Teitelman 78;p.17.16], but the user can just use the intuitive definition of "looks like" provided here.

If the respelling set consists of exactly one respelling, and the two words look very much alike,[9] ***Affirm*** automatically corrects the misspelling to the respelling, and reports the correction to the user.

If the system cannot make a good guess of the word that was meant, then the user is asked to choose a word from a small list of possible respellings. The user can reject all of the list by typing a slash, which in this context means "none of these choices". At this point, the user is asked to correct the misspelling using the entire spelling list of possible choices (which may not be small). This normally suffices, but again the user may type a slash, in which case the context of the particular misspelling determines further action on ***Affirm***'s part. For example, the default command name when the user refuses to correct a misspelled command is note, the comment command. In other contexts, an error is generated, and control returns to the executive.

The default action taken upon user rejection of all choices in a particular context is discussed in the appropriate portion of this manual.

In the example below, the user mis-typed a command word. The system computed a small set of possible respellings (very small: only two choices). The user typed a question mark to see the choices, and decided to reject both by typing a slash. ***Affirm*** then gave the user the entire list of possible command names to choose from. After perusing this list, the user again rejected all choices. The default action in this case was to choose the default command name, note. User-supplied input is shown in a **bigger font** than system-typed output.

[9] I.e., they match letter for letter at least 60% of the time. This heuristic parameter is accessible to the user via the profile entry *DontAskJustTake*.

```
pleutt;                                                          user goofs
what?  Please correct pleutt using the list: let or put.         system responds
Please correct pleutt: ?                                         user asks for choices
one of:
/ (None!), let, put,
Please correct pleutt: / (None!) [confirm] (cret)                User says "None"
                                                                 system now asks the
                                                                 user to retype the
                                                                 mis-understood word
No, eh?  Please retype pleutt: ?                                 user asks for choices
one of:                                                          system displays lots
                                                                 of choices
/ (None!), ;, @, abort, adopt, Affirmed?, annotate, apply, arc,
assume, augment, axiom, cases, choose, clear, compile, complete,
declare, define, denote, discard, down, e, edit, employ, end,
enter, eval, exec, fix, forget, freeze, genvcs, gripe, help,
infix, interface, invoke, keep, let, lisp, load, name, needs,
next, normalize, normint, note, ok, print, profile, put, quit,
read, readp, redo, renumber, replace, resume, retry, review,
rulelemma, save, schema, search, set, split, stop, storage,
sufficient?, suppose, swap, thaw, theorem, transcript, try, type,
undo, up, use, axioms, interfaces, rulelemmas, schemas
Please retype pleutt: / (None!);                                 user rejects all
Using 'note'; skipping to semicolon...                           system default action
                                                                 for command names
```

3.4. The Event History and the Undo, Redo, and Fix Commands

Affirm utilizes the history mechanism provided by Interlisp to provide the user with a limited undo capability. Almost all commands that modify system state can be undone, returning the system to the state it was in before the command was processed. ***Affirm*** automatically undoes the partly-completed effects of commands interrupted by errors. The effects of a certain number[10] of previous commands are saved by the system in a *history window*; this structure can be thought of as a window of the most recent events in the complete history of events performed by the system. When the user types

    undo *eventNumber*;

the effects of the command at *eventNumber* are undone, as long as the event associated with the event number is still remembered in the history window. Unfortunately, undoing commands out of strict reverse order of processing quickly leads to fatal (or at least highly curious) results. The user is thus enjoined to only undo commands *in strict reverse order of processing*. Typing

    undo;

without an explicit parameter defaults to the most recent event that completed without an error

[10]The number of previous events saved is the value of the user profile entry *HistoryWindowSize* and thus can be set by the user.

(because erroneous events are automatically undone).

Any event still remembered in the history window can be redone simply by typing

```
redo eventNumber;
```

A badly garbled command can often be fixed via the fix command. The fix command places the text of the command to be fixed into a text editor determined by the profile entry *TextEditor* [§3.13]. After the user is finished editing the command line, the command is read back in by ***Affirm*** and processed. The user can abort the fix, so that the command will not be processed.

```
fix eventNumber;
```

processes the command at event *eventNumber*, while

```
fix;
```

with no explicit parameter defaults to the previous command.

The history mechanism consumes some amount of list space that may be required for proof attempts during long sessions. If the space gets low, the history window can be undoably purged by using the discard history command, and the size of the window can be adjusted by modifying the value of the profile entry *HistoryWindowSize*. See the Users Guide for guidance.

The read command [§9.6], which reads a sequence of ***Affirm*** commands from a file, is considered *one* event by the history mechanism. This means that the one event may be quite large, thus consuming a great amount of list space. Again, the discard history command and the profile entry *HistoryWindowSize* are useful here.

Affirm begins a new session with about 100 free pages of memory not allocated to a specific function. As the user specifies types, proves theorems, etc., the free space is allocated to various uses. Once the unallocated space is totally allocated, the system is in danger of running out of space, when the Interlisp garbage collector tries to obtain more space for a specific use. At this point, the system prints

```
STORAGE FULL
NIL
```

on the user's terminal, and then aborts the command currently being processed. Don't totally panic! There is still enough space to save the appropriate data type specifications, print the proof trees, list the propositions, etc. But there is not enough space to continue the specification or proof attempt.

Affirm will automatically warn the user when the number of free core pages not yet allocated by the Interlisp garbage collector first drops below ten. The test is made at the beginning of each command cycle, just before the "U:" prompt is printed. The user can still perform much useful work after the warning is printed. But if there is still lots to do, the point of the space warning is a good place to perform a checkpoint. If space is a problem, the user should compile all stable types [§9.6], reduce the size of the history window (using the profile entry *HistoryWindowSize* [§3.13]), and should use the storage command (as described below).

---

fix [*eventNumber*];
: places the user in a text editor (determined by the profile entry *TextEditor*) with the text of the command issued at event *eventNumber*. The default event when *eventNumber* is not explicitly supplied is the previous event.

discard history;
: purges the history window. This command can be undone.

print history;
: prints the user-issued commands still resident in the history window.

storage *degree*;
: *Degree* is one of {normal, severe, tight}. The storage command provides a small amount of control over the page allocation mechanism of the Interlisp garbage collector. At present, we only suggest its use when ***Affirm*** explicitly warns the user that space is low. The user should then type

    storage severe;

redo [*eventNumber*];
: re-executes the command at event *eventNumber*.

undo [*eventNumber*];
: undoes the effects of execution of the command at event *eventNumber*, if possible.

3.5. Error Processing

There are essentially three sources of errors detected by ***Affirm***: internal code errors, user interrupts and user-supplied input. Each is described in turn below.

3.5.1. Internal Code Errors

The subject of internal code errors is a rather difficult one to deal with in a reference manual. First of all, we do not claim to detect all internal errors. The user will undoubtedly be hard-pressed to determine whether an error message is coming from ***Affirm***, or from the underlying Interlisp system. Most messages from ***Affirm*** answer the obvious question "Is it safe to continue?", either by explicitly stating it, or by forcing a halt, or by continuing automatically, as the individual case dictates. Most Interlisp errors cause the system to halt. Whether or not it is safe to continue (by typing ↑, or `(AffirmExec)` -- see the Users Guide) becomes a moot point; no one will believe the proof anyhow.

3.5.2. User Interrupts

The Interlisp system underlying the implementation of ***Affirm*** provides a plethora of control characters, each of which could conceivably be useful in a particular situation. For the most part, however, only a small subset of the thirty-odd control characters are useful. This set includes ↑A, ↑C, ↑D, ↑E, ↑F, ↑K, ↑N, ↑Q, ↑R, ↑T, ↑X, ↑Z, escape, and DEL. Unfortunately, the meaning of most of these control characters is not consistent between ***Tops-20*** and ***Tenex***, the two operating systems under which ***Affirm*** runs. Section III.6 discusses the meaning of each control character under each operating system. For the most part, users should avoid typing most control characters. Exceptions to this general rule are noted below.

3.5.2.1. control-E: Command Abort

control-E aborts the command currently being processed, automatically undoing any effects it may have had, and returns the user to the ***Affirm*** executive. If the user types several **control**-E's in a row quickly when the system is heavily loaded, the undoing may itself be aborted. In this case, ***Affirm*** tells the user. An explicit <u>undo</u> command should then be issued.

control-E can be hit anytime, with two exceptions:

- While the user is typing the text of a <u>gripe</u> command [§3.8], **control-E** is treated like any normal character (i.e., text); and
- Inside some text editor called as a result of a <u>fix</u> [§3.4] or <u>review</u> command [§3.10.3], **control-E** has whatever meaning assigned to it by that particular text editor.

In certain instances, **control-E** will <u>not</u> undo <u>everything</u> done since the last user-issued command. In particular, the Auto Mechanism [§3.12] can be aborted, without undoing the command that caused the Auto Mechanism to be invoked. The system prints a separate message if the user types **control**-E when the Auto Mechanism is running. If the desired effect is to abort and undo the command, the user should next issue the <u>undo</u> command.

3.5.2.2. control-D: Panic Abort

control-D is an Interlisp abort which simply returns to the top-level Interlisp executive (colloquially termed <u>EvalQuote</u>). Once there, the user can get back into ***Affirm*** by typing

(AffirmExec)

The global data structures modified by commands issued before the **control-D** will not be reset. Thus, everything done up to the command that was being processed when the **control-D** was typed, will still be remembered. Unlike **control**-E, no automatic <u>undo</u> is performed when **control-D** is typed.

For the most part, **control-E** <u>should</u> be sufficient. **control-D** should be used only in cases where **control-E** seems to have no effect. The only case we know of involves <u>read</u>ing files containing Interlisp code (that should be <u>load</u>ed instead [§9.6]).

3.5.2.3. control-C, control-F, and control-T: Operating System Functions

control-C aborts ***Affirm***, returning the user to the operating system executive. Normally, a **continue** command to the executive will return to ***Affirm*** with no ill effects (except that the input buffer is cleared by the operating system).

control-F and **Escape** are useful when ***Affirm*** requests a file name. They have the same meaning as in the operating system executives. In fact, when ***Affirm*** asks for a file name, it actually uses the operating system to obtain it. Thus the normal file name protocol is in effect. ***Affirm*** has a set of file name conventions; various commands have default file extension fields that will be used if the user does not explicitly fill in the extension field. The file name conventions are documented in Section 9.5.

control-T is intercepted by Interlisp, which then prints a one-line summary of what functions are currently running, along with the system load average. It may very well be the character hit most often (perhaps after carriage return and space). It has no effect on ***Affirm***, but does soothe the user's nervous system!

3.5.3. Errors in User-Supplied Input

There are several categories of error under the general classification of user-supplied input. These include syntax errors, spelling errors, type mismatches, and undeclared variable and operation names.

- Syntax errors: The input parser of the system is a recursive descent parser employing backup. This severely limits our ability to provide reasonable error messages, much less recover from syntax errors. The user is generally forced to try again, although the fix command can be quite useful [§3.4].

- Spelling errors: ***Affirm*** automatically corrects misspelled identifiers in certain contexts. If it cannot correct the misspelling, ***Affirm*** usually assumes the identifier is an undeclared variable or operation name.

- Type mismatches: Type mismatches occur when one or more parameters to an operation are not of the expected type. ***Affirm*** detects and reports the error. The user is generally forced to re-type the command. Again, the fix command is quite useful in this context for editing the command line.

- Undeclared references: The system recursively enters a *lower executive*, as is described below.

3.5.4. The Lower Executive

Whenever ***Affirm*** processes a command line containing a variable or operation reference for which a prior declaration does not exist, processing is temporarily suspended, and a *lower executive* is entered. The user is then asked to declare the variable or provide the interface declaration for the operation. The lower executive provides the user with the full set of commands. Thus the user can issue any command, not just the declaration commands. Further references to undeclared variables or operations drop the user into successively lower executives. The command prompt is enhanced for each lower executive to provide the depth of nesting; an attempt is made to provide the user with a feeling that he or she has truly suspended processing of one command. After the requisite declarations have been typed, the user can type the ok command to resume processing at the next higher executive.

If however the user wishes to abort the suspended processing altogether, the user can type the abort command in place of the ok command. This command returns control to the next higher executive. The user can then use the fix command (with an explicit parameter) to fix the errant command line.

---

abort; returns the user to the next higher ***Affirm*** executive (if there is one), and aborts the suspended command. The command can then be fixed, or forgotten.

ok; returns the user to the next higher ***Affirm*** executive (if there is one), and resumes processing of the suspended command. If this command still has errors in it, the user may well be placed into a lower executive once again. The abort command is useful here, too.

stop; returns the user to the command level from reading a file. ***Affirm*** next expects input from the terminal. The stop command should be the last command of a sequence of commands in a file. It should also be used if the user wishes to abort reading from a file. If the read is interrupted (either by detection of an error, or the user typing **control-E**), the user can type stop to abort the remainder of the file processing.

3.6. The Help System

The Help system is intended to provide an online reference manual which may be easily queried. At present, we have only a rudimentary skeleton of what we hope to evolve. The user can ask for information about the available commands.

The help command takes a *subject* as a parameter, where currently a subject is simply a command name. A short paragraph is displayed which attempts to explain the syntax, semantics, use, or meaning of the requested subject. Currently, lots of pointers back to the user's desk copy of the ***Affirm*** reference manual are provided.

help [*Topic*];

lists information about *topic*, if any information is available.

3.7. The News Facility

When a session first begins, ***Affirm*** automatically displays any news about the system that the user has not yet seen. The news items are displayed only once, using the time of day that ***Affirm*** was last run in the user's login directory as the determining factor. Thus any system news added to the database while the user is running ***Affirm*** will be seen the *next* time that user uses the system.

3.8. The Gripe Facility

The gripe command provides a mechanism for sending suggestions or possible bugs to the people maintaining ***Affirm***. The ***Affirm*** system is very big, and as with most big systems there are many problems. If a user is having a problem or has a suggestion, he or she should type

gripe *subject*;

where *subject* is a very short description of the problem or suggestion (it must be present, but must be less than 35 characters long). The user will then be asked to type a message explaining the problem in greater detail (which can be any number of characters in length). The usual editing control characters are available while typing the message, but **control-E**, the command abort character, does not work. The message is ended by typing **control-Z**. The user is then given the chance of either sending the message or aborting the command. The system will next ask if the user wants to send the current transcript as another message. Normally the transcript is not required to fix the error, but novice users should send the transcript.

All the mail for gripe goes to the message file in directory <PVREPORT> on ISIF; all users are encouraged to look through the file to see what sorts of problems we are aware of. As we process gripes, the documentation of the problem is moved into file <Affirm>KnownBugs.TXT, a message file that users are encouraged to view using their favorite message system.

gripe *subject*;

creates a message to be sent via the ARPANET to ***Affirm*** maintenance personnel. The system will ask the user to type the body of the message, which is terminated by **control-Z**. After the message is completed, the user has the options of sending the message, or aborting the gripe. The transcript [§9.2] can also be sent along as a separate message if it is pertinent to the documentation of the problem or suggestion. (It is usually not necessary.)

3.9. The Timing System

This mechanism is quite simple, and just prints out the number of CPU seconds used by each command. It is activated by setting the profile entry *Timer* to On, and de-activated by setting *Timer* to Off. The timer is smart enough to correctly deal with file reading, by providing the timings for each command in the file, as well as the sum total of the CPU time of the read command.

3.10. Miscellaneous Commands

Several commands that do not neatly fit into any executive submachine are described here.

3.10.1. The note Command

The user can introduce comments into the transcript of the session via the note command.

---

note *arbitraryTextExceptSemicolon*;

The text is placed in the transcript. No other processing is performed.

3.10.2. The transcript Command

Affirm automatically keeps a *transcript* file which is a nearly verbatim echo of all input and output generated during a session. This file is named according to the profile entry *TranscriptFileName* [§3.13]. The transcript command can be used to turn off the transcript, to turn it back on, or to switch to a new transcript file. The command is fully explained in Section 9.2.

---

transcript;

causes the old transcript file to be closed and a new one opened. The name of the new file will be the same as the old one (but will be in the connected directory, if there is one); the version number will be incremented [§9.2].

3.10.3. When the screen gets full: the review command

The review command reminds the user of the name of the current transcript file and places the user in an editor.[11] Thus the transcript can be reviewed by using editor commands. This is quite useful if, for example, the proposition being proven is so large that, using a CRT terminal, the whole proposition will not fit on several screens. The HP2640A's at ISI have a memory that allows

[11]The particular editor is determined by the profile entry *TextEditor* [§3.13].

approximately three screen-pages of 24 lines each to be scrolled through. But propositions larger than that can prove too unwieldy to manipulate. To return to ***Affirm***, perform the editor's quit command. ***Affirm*** will continue the transcript in the *same* version of the same file.

---

review;

places the user in a text editor determined by the profile entry *TextEditor*, with the transcript file. The user can then use editor commands to review the events in the file. Each command begins with "U:".

3.10.4. The thaw command

This command is the opposite of the freeze command. It takes one parameter, the name of a file containing a session frozen by a freeze command.

Most users will not ever have a use for this command, since the frozen session can be started in ***TOPS-20*** or ***Tenex*** simply by typing the file name at the operating system executive level. The command was added primarily for the MIT-***Affirm*** group.

---

thaw *fileName*;

file *fileName* contains a previous ***Affirm*** session frozen by the freeze command. That session is continued. *Not needed by the normal user.*

3.10.5. The print Command

The print command displays data type specifications, proof status, individual proof steps, etc, on the terminal. There are a large number of options to this command, most of which have to do with printing types or propositions or proofs. The complete set is documented here, with pointers to the appropriate sections of this document for explanations of the as-yet undefined terms.

---

print ?;

displays a list of all the keywords that can follow the command word print. Equivalent to the command

print known PrintObjects;

print assumptions;

prints the propositions used as lemmas in the proof attempt of the Current Theorem [§7.2], if the lemma has a proof status [§7.2] of assumed.

print BadEquations;

lists the rules that have been suppressed during the Knuth-Bendix [§5.2] convergence test, if any.

print both;
Like print proof but lists all the propositions in the proof tree. *Verbose and rarely useful.*

print IH;
prints the definition of each of the inductive hypotheses [§7.4.2.3] in the Current Proposition (if any).

print file *fileName*;
FileName may contain **escapes** and **control-Fs**, which are interpreted by the operating system in the normal manner, although the file name expansion does *not* occur as the file name is typed. [§9.5] [§3.2.1]

print known *ObjectName*;
prints the names of all known elements of the object class. The object classes as of Version 1.21 are AffirmObject, arc, axiom, command, definition, directory, file, fileType, interface, lemma, node, printObject, profileEntry, schema, type, typePart, and variable. Not all of these make sense in the context of print known.

print [parts *TypeParts*] [types *typeNames*] lhs *expression*;
This feature provides the rudimentary capability of listing those rules that match some *pattern*. *TypeParts* is a list selected from the set {axiom, lemma, defn, schema}; the list may be empty, in which case the default value axiom is supplied. *TypeNames* is a list of type names; the list may be empty, in which case the keyword need not be typed. The default value is the list of all currently defined types. *Pattern* is an expression, restricted to one of two simple forms: operator, or operator1(operator2).

The search mechanism works as follows. Each rule in the requested set of types that is a member of one of the requested parts is pattern-matched against the pattern; if it succeeds, the rule is listed. If it fails, the rule is ignored. The pattern-match process is as follows. Only the left-hand side of a rule is used. If the pattern is a simple operator, a match succeeds if the *main* operator of the left-hand side is this operator. If the pattern is of the form operator1(operator2), then operator1 is the main operator, and operator2 is *any internal* operator. If the left-hand side of a rule has operator1 as its main operator, and contains a reference to operator2 as an internal operator, the match succeeds.

For example, the command

```
print lhs join(apr);
```

will list all the axioms whose left-hand-side main operator is *join*, and which also reference the operator *apr* as an internal operator. (This example is useful for the type SequenceOf*X*, for quite a few types *X*.)

print next;
lists the proposition that would become the Current Proposition if the user issued the next command [§7.6.1.4].

print proof *theoremNames*;
lists the proof trees [§7.2] for the indicated theorems.

print proof theorems;
lists the proof tree [§7.2] for all theorems (named or not).

print prop *propositionNames*;
lists the proposition associated with each of the indicated names.

print result;
simply re-lists the Current Proposition [§7.2]

print status;
lists the current status [§7.2] of each theorem.

print type *typeName* [*typeParts*];
lists the type *typeName* [§4.5.1].

print uses;
lists the dependencies of each theorem on any other it uses as a lemma [§7.4.1.1].

print variables;
lists the universal and existential quantifiers of the Current Proposition [§6.2.2] [§7.2].

3.10.6. Operating System Interfacing Commands

---

exec; invokes the operating system executive as a subroutine. The user can do anything that can be done at the original executive without destroying the files and memory associated with ***Affirm***. To continue with the ***Affirm*** session, the user should type POP at the operating system executive command level.

quit; stops ***Affirm***, returning to the operating system executive. The user can return to ***Affirm*** by immediately typing CONTINUE at the operating system executive command level.

3.11. The Profile Mechanism

Associated with each user is a *profile* of about fifty separate entries, providing the user with some control over various displays and their formats. This database of information is kept in the user's directory in a file named "--AffirmUserProfile--".

The initial values of the profile entries are established when the user begins a session by first defaulting each entry [§3.13], and then reading the user's profile file. Any entries not in the profile file thus keep their default values. The user can directly modify any entry, read and write other profile files, and enter a query mode which displays each entry and accepts new values. There are two main modes of enquiry and modification of profile entries:

1. a *profile dialogue*, modelled after that of ***XED***[12], in which the system displays each entry

[12]A local text editor.

(grouped by family), and prompts the user for a new value; and

2. *direct manipulation*, where the user provides a sequence of entry names, possibly followed by desired new values, and the system processes each element of the sequence, interacting with the user as necessary.

The best way to get a feeling for how the profile mechanism works is to try it out. Typing

```
profile;
```

causes the system to run through the dialogue. Each time the system asks a question, the user can type a question mark to obtain the range of responses.

3.11.1. Direct Manipulation

The second mode of use of the mechanism, the direct manipulation mode, is employed by providing a series of parameters to the profile command. The profile command's parameter list is composed of a series of transactions, separated by commas and/or the noiseword and; the list is terminated by the customary semicolon. A transaction consists of at least a profile entry name, in any casing whatsoever. If that is all there is, then the transaction is termed a query, and the system will simply display the current value of the entry. The entry name can also be followed by a question mark to explicitly represent the fact that the current value of this entry has been requested. If the entry name is followed by an equal sign and an identifier or integer, then it is taken to be the new value of the entry. This type of transaction is termed a set. Naturally, error checking is performed on both the entry names and the atoms representing new values.

Some examples of input (the user should try these to see the output):

```
profile TerminalLineWidth = 98, and LessOutputDesired = On;

profile showrules = true and tERMINALlINEwIDTH?,
    lessoutputdesired ?;

profile RuleLHSPercentage = 49 and ShowRTULES = no;
```

The user can also set several entries to be the same value via a multiple-assignment-like statement:

```
profile LessOutputDesired = ExpertUser = ShowRules = Yes;
```

In this case, all three entries are turned On.

3.11.2. Boolean-Valued Entries

Most profile entries have Boolean values. For readability, we define the set of possible values of Boolean-valued profile entries to be {On, Off, Yes, No, True, False}, where On, Yes, and True are equivalent, and Off, No, and False are equivalent. (This idea was unabashedly borrowed from

***SCRIBE*.)**[13] This set is often referred to as OnOffValues.

3.12. The Auto Mechanism

Commands quite often come in sequences: after a readp, quite often the next command issued is genvcs. After an employ, if the basis step is immediately proved, the first thing the user does is invoke IH. If there is an embedded if-expression, and the system says "(The cases command is applicable)", the user types cases.

The mechanism in ***Affirm*** that oversees automatic performance of common command sequences is called the *Auto Mechanism*. It is controlled by a series of profile entries. There are presently 12 different actions that can be "automated" in certain contexts. Each profile entry in the Auto Mechanism family can take the values described below.

On, True, Yes
: The normal On values.

Off, False, No
: The normal Off values.

Tell
: Equivalent to On, but an extra message is printed notifying the user something is about to happen.

Ask
: The system will stop and ask the user if the automatically-applied command should occur. Expected responses are Yes or No.

In the beginning, it is suggested the user set the values of desired members of this family to Tell, to become familiar with their behavior. Then later set the value to On, Off, Tell, or Ask, as is appropriate.

The 12 profile entries are named *AutoAnnotate*, *AutoCases*, *AutoFreeze*, *AutoGenvcs*, *AutoInvokeIH*, *AutoNext*, *AutoNormint*, *AutoPrintProof*, *AutoPrintProofTheorems*, *AutoReplace*, *AutoSearch*, and *AutoSufficient*. They are described in the section containing all the defined profile entries.

We recommend arming (setting to On, Ask, or Tell) profile entries *AutoAnnotate*, *AutoCases*, *AutoFreeze*, *AutoGenvcs*, *AutoNext*, *AutoNormint*, *AutoPrintProof*, and *AutoPrintProofTheorems*. We also recommend that *AutoReplace* be left Off for normal theorem-proving.

[13] ***SCRIBE*** is a document preparation system developed by the Computer Science Department at Carnegie-Mellon University, and now distributed by UNILOGIC, Ltd., Pittsburgh.

3.13. Currently Defined Profile Entries

The current profile entries are listed in alphabetical order below. For each entry, we discuss its purpose, its possible values, its default value, and possibly give some pointers to other parts of this document.

AnnotatingTranscript

Possible values: OnOffValues. Default value: Off. Causes user-typed commands to be surrounded with ***SCRIBE*** commands, effectively putting user commands in a different font when the transcript is run through ***SCRIBE***. This mechanism is used to create the annotated transcripts used as examples throughout the Users Guide and Annotated Transcripts Volumes of the Reference Library. [§9.2]

AutoAnnotate

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the proof of the current theorem is complete. If it is armed, then an annotation is written: "*Status* by *User* using Affirm-*Version* on *date* in transcript *transcriptFileName*", where *status* is either proven or assumed. [§7.9.3]

AutoCases

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the Current Proposition contains embedded if-expressions. If it is armed, the cases command is performed. [§6.3.3]

AutoFreeze

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the user quits a session. If it is armed, the freeze command is performed; the freeze file name can be supplied as the parameter of the quit command which invoked this auto-command. If no file name is supplied with the quit command, the system will stop and ask the user for a file name. If the user types an escape, the default file name (from the profile entry *FrozenFileName*) is used. [§9.6]

AutoGenvcs

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable immediately after the readp command. If it is armed, the genvcs command is performed, with the list of newly parsed Pascal program unit names as its parameter. [§8.3]

AutoInvokeIH

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the Current Proposition contains references to IH. If it is armed, the invoke command is performed, with parameter IH [all]. [§7.5.2.3]

AutoNext

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the proof of a branch is completed. If it is armed, the next command is performed. [§7.6.1.4]

AutoNormint

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the Current Proposition contains arithmetic expressions. If it is armed, the normint command is performed. [§6.7]

AutoPrintProof

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the proof of a theorem is complete. If it is armed, the print command is performed, with parameter proof. [§7.7.4]

AutoPrintProofTheorems

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the user quits a session. If it is armed, the print command is performed, with parameters proof theorems. [§7.7.4]

AutoReplace

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the Current Proposition contains equalities. If it armed, the replace command is performed, with no parameters. [§7.5.2.2]

AutoSearch

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable when the Current Proposition contains existential quantifiers. If it is armed, the search command is performed. [§6.6]

AutoSufficient

Possible values: {On, Off, Ask, Tell}. Default value: Off. This profile entry is applicable after the end command. If it is armed, the sufficient? command is performed, with the name of the type just closed as its parameter. This determines whether or not the type is *recognizably sufficiently complete*. [§4.4]

AxiomGrouping

Possible values: OnOffValues. Default value: On. When listing a data type, should the axioms of the type be grouped together, as in

```
axioms
        null join s = s,
        (s1 apr i) join s2 = s1 join (i apl s2);
```

or should they be listed individually, as in

```
axiom  null join s = s;
axiom  (s1 apr i) join s2 = s1 join (i apl s2);
```

The former may be prettier, but the latter is much safer when reading a data type definition from a file. **control-E**, the command-abort symbol, essentially causes a skip to semicolon; if one of the first few axioms in a long list of axioms causes problems during the rewrite rule addition process, when the user aborts the one axiom, the remaining list is forgotten, too. The best way to avoid this is to keep the lists quite short. [§4.2.3]

BreakAccess

Possible values: OnOffValues. Default value: Off, no breaks allowed. Should the user be put into an Interlisp break if the system tries, or be kept in ***Affirm***? [§III.4]

CautiousCompletion

Possible values: OnOffValues. Default Value: Off. If this entry's value is On, whenever a new rule is added to RuleSet, whether by the user [Chapter 4] or by the unique termination algorithm [Chapter 5], the user is asked for confirmation.

DefineGrouping

Possible values: OnOffValues. Default value: On. Controls the listing format of definitions. See *AxiomGrouping*. [§4.2.3]

DontAskJustTake

Possible values: any integer between 0 and 100. Default value: 40 (percent). The value represents a percentage relative agreement value used by the ***Affirm*** spelling corrector. The spelling corrector tries to *match* the misspelled word against a set of possibilities, by iteratively computing *closeness*, and attempting to reduce the size of the set of close matches. This iteration stops when the set of close matches gets below a specified size. If it turns out that only one possible respelling remains, then the spelling corrector must decide whether or not to *assume* the misspelled word is meant to be the one possibility, or to ask the user to *confirm* it. If the percentage closeness is greater than the value of the entry *DontAskJustTake*, then the user is not queried; the respelling is assumed. Otherwise, the user is asked to confirm the respelling. [§3.3]

EnquireAfterFreeze
Possible values: OnOffValues. Default value: Off. Similar to *EnquireInitially*, this entry determines whether or not the user profile will be *reinitialized* upon startup of a system previously saved by the freeze command [§9.6]. If Off, the previous values of the profile will be retained.

EnquireInitially
Possible values: OnOffValues. Default value: Off. Should the profile mechanism start the user in a profile enquiry dialogue as soon as ***Affirm*** is entered?

FreezeFileName
Possible values: any valid file name. Default value: directory = connected, name = Frozen-Affirm, extension EXE on ***Tops-20*** and SAV on ***Tenex***. This entry provides the default file name for the freeze command when the user does not explicitly provide one. [§9.6]

GarbageCollectionMessage
Possible values: {Normal, None, Compact}. Default value: None. Allows more control over the format of the garbage collection message. See the Users Guide.

GarbageCollectionPages
Possible values: any positive integer. Default value: 40. If the number of free pages drops below this number, the user gets a message extended with "...*number* pages left!", if *GarbageCollectionMessage* is Normal or Compact.

HistoryWindowSize
Possible values: any integer greater than or equal to three. Default value: 30. The number of events saved in the history, for the fix, redo, and undo commands. [§3.4]

InterfaceGrouping
Possible values: OnOffValues. Default value: On. Controls the listing format of interface declarations. See *AxiomGrouping*.

LemmaGrouping
Possible values: OnOffValues. Default value: On. Controls the listing format of rulelemmas. See *AxiomGrouping*.

LessOutputDesired
Possible values: OnOffValues. Default value: Off. When this entry's value is On, CurrentProposition will not be printed before normalization. [§6.3.2]

ReadAnotherProfileFile
Possible values: OnOffValues. Default value: On. At system startup, the system automatically reads the user's initial profile file. If that file says to read another one, it does. If *that* file says to read yet another file, it does, ... etc.. This entry's value says whether or not to read another file, and is used in conjunction with the profile entry *UserProfileFileName*, which contains the file to

be read. Cycles are noticed and avoided. The default value of On may seem strange, but the value of *UserProfileFileName* is set to the user's connected directory, and the initial profile file is read from the user's login directory. Thus the default profile file is read from the login directory, which defaults to another read from the connected directory. If the login directory is identical to the connected directory, the profile mechanism notices the cycle and halts. If only the login directory's profile file is desired, the user should turn *ReadAnotherProfileFile* Off.

SaveOnlyChangedEntries
Possible values: OnOffValues. Default value: On. When the profile is saved, should only those entries that differ from their default values be saved, or should all entries be saved?

SchemaGrouping
Possible values: OnOffValues. Default value: On. Controls the listing format of schemas. See *AxiomGrouping*.

ShowNormint
Possible values: OnOffValues. Default value: Off. This profile entry, when On, traces the actions of the normint command. Useful every once in a while when the normint command performs a simplification the user cannot follow. Just undo the normint event, turn the *ShowNormint* profile entry On, redo the normint event, and turn *ShowNormint* back Off. [§6.7]

ShowRules
Possible values: OnOffValues. Default value: Off. When this entry's value is On, each application of any rewrite rule will be reported [Chapter 5]. This creates quite a lot of output, but can be quite educational in understanding what ***Affirm*** does to a proposition during normalization. [§6.3.2]

ShowRuleSimplification
Possible values: OnOffValues. Default value: Yes. The first step performed in the procedure which adds new rewrite rules to the rewrite rule database is to simplify the new rule, using the current rewrite rules. Should the system display the simplified rule? [§5.2]

TerminalLineWidth
Possible values: any number between 20 and 132. Default value: 79. Our CRT's are 79 characters wide. To nicely fill a printed page when annotating, the user is advised to set the terminal line width to 88 (determined via much experimentation).

TextEditor
Possible values: {XED, SOS, TECO, EMACS, RMODE, TED, POET}. Default value: XED. When the user issues the review or fix commands, what editor should be invoked?[14]

Timer Possible Values: OnOffValues. Default value: Off. This entry turns on or off the timing of each command (in CPU seconds). [§3.9]

TranscriptFileName
Possible values: any valid file name. Default value: directory = login, name = AffirmTranscript, extension = date in dd-mon-yy format. This entry provides the default name of the session transcript both at the time the session begins, and in the transcript command when the user does not explicitly provide a file name. [§9.2]

TypesInInterfaces
Possible values: {Types, Variables}. Default value: Variables. When listing the interface

[14] ***Affirm*** does not support SOS line numbers

declarations of a data type, should the parameters be variable names, or type names?[15]

UserProfileFileName

Possible values: any file name, using the normal operating system conventions. Default value: directory = connected, file name = --AffirmUSERPROFILE--, extension = empty. This entry is used in conjunction with the profile entry *ReadAnotherProfileFile*; this entry's value tells what file to read. This entry is also used to determine the name of the output file for profile dumping when the user responds to the "file name?" question of the profile enquiry with an escape.

UsingTed

PossibleValues: OnOffValues. Default value: Off. This profile entry makes ***Affirm*** behave as the ***ATED*** interface expects it to. This is not something most users should worry about: the MIT-***Affirm*** group needs it. Now new versions of ***Affirm*** can be used immediately with ***ATED***, and should not require extra changes.

[15]***Affirm*** cannot currently read in a type specification where the interface declarations contain type names in the parameter positions.

4. The Specification Machine

4.1. Introduction

The ***Affirm*** Specification machine builds data structures used by the Logic, VC Generation, Rewrite Rule, and Theorem Prover machines. Internally the Specification machine maintains a TypeSet, a LocalDeclarationSet, an InterfaceSet, a RuleSet, and a ContextStack. The objects in these data structures are as follows:

type a record structure with fields (type name, set of local declarations, set of interfaces, set of rules)

local declaration
: a record structure with fields (variable name, type name)

interface
: a record structure with fields (function name, infix switch, domain type list, range type)

rule a record structure with fields (rule class, left expression, right expression)

context
: a type name

4.2. Types

A type is a four-tuple (type name, set of local declarations, set of interfaces, set of rules). The set of all type specifications is TypeSet. The system provides several predefined types, most notably *Integer* and *Boolean*. Other elementary types are stored in files in the ***PVLIBRARY*** directory. The user may specify new types by the command

type *typeName*;

which creates a new element of TypeSet with a declaration of a dummy variable in the local declaration set, an equality operation in the interface set, and the reflexive axiom of equality in the rule set. The command

edit *typeName*;

allows the user to modify an existing type specification. Any specification commands given after a type or edit command is performed in the context of this type. The end command closes the current type specification. The data structure keeping track of the open types is the stack ContextStack. The top element is the type currently being edited; the end command simply pops this stack, while the type and edit commands push a new element onto it.

---

type *typeName*;
: specifies *typeName* as the name of an abstract type, whose specification will be given by subsequent commands. The name *typeName* is added to the TypeSet and is pushed onto ContextStack. If *typeName* is already a member of the TypeSet, its existing specification ***will be discarded***. Each new type is automatically provided with one variable declaration (the name of

which is controlled by the profile entry *DummyVarName*), a declaration of an equality operation, and an axiom explicitly stating that the equality operation is reflexive. (The remaining properties of an equality operation are *assumed*, and should be validated by the creator of the type.)

edit *typeName*;
: *typeName* must be a member of TypeSet. *typeName* is pushed onto ContextStack, thus making the local declarations of *typeName* available for referencing.

end; causes ContextStack to be popped, ending the current type's specification and returning to the previous context. (If this is the only entry in ContextStack, nothing happens.)

print known types;
: displays the currently defined type names.

4.2.1. Local Declarations

The local declarations or *variables* of a type are simply abstract values of the type. The elements of the local declaration set of a type are pairs (variable name, type name). The variable name is an identifier and the type name must be a member of TypeSet. The command

declare *id*: *typeName*;

where *id* is a variable name and *typeName* is a member of TypeSet, is used to add new elements to the local declaration set. During type specification only the variable names of the type currently being specified are available to the user. The command

adopt *typeName*;

enables the user to copy the variables of the local declaration set of the previously defined type *typeName* into the CurrentContext. Primed variables of the adopted type, usually only generated by ***Affirm*** [Chapter 6], are not copied. If a variable name of an adopted type is the same as a variable name already declared in the current type and their respective range types are not identical, then the adopted variable name is extended with a dollar sign character $.

Variable names and interface names (see below) in the same type *must be distinct*.

---

declare v_1, ..., v_n: *typeName*;
: each of the v_i is declared to be a variable of type *typeName*. *typeName* must be a member of TypeSet. Each of the declarations is added to the local declaration set of the current type. *TypeName* can also be the upper-case letter T, which denotes the current type (the entry on the top of the ContextStack).

```
declare q, q1: SequenceOfElemType;
declare x: ElemType;
```

adopt *typeName*;
: sometimes it is necessary to prove theorems about operators which are associated with types other than the current one. The operators of the type will be referenceable, because the type is

in TypeSet. However, the variables of that type may not be referenceable in the current context. Rather than enter the necessary variable declarations manually, the adopt command provides a convenient way to *copy* all the non-primed declarations of a type over to the current one. Should any name conflicts occur, the variables being copied will be renamed by appending dollar sign characters ($) to them.

adopt SequenceOfElemType;

discard variable *variableName*$_1$ [, ... *variableName*$_n$];

discards the variables *variableName*$_i$ for *i* from 1 to *n*, from the current type. Note that any use of the variables, such as in interface declarations or rules, is now undefined, and may be inconsistent. The system does not presently check for this condition. (However, the user will certainly feel the effects later!) It is the user's responsibility to discard or redefine interfaces and rules referencing the newly-discarded variables.

4.2.2. Interfaces

The interface command declares the domain and range information--the interface--for operations of the type being specified. Each interface entry is a four-tuple (function name, infix switch, domain type list, range type). The elements of the domain type list and the range type are members of TypeSet. Note that the domain type list of a constant function is empty. The infix switch determines whether expressions involving the function should appear in prefix or infix form when printed by ***Affirm***. New elements are added to the interface set by the interface command. The command

interface *functionName*(*x1*, *x2*): *typeName*;

adds an element to the interface set of the CurrentContext type of the form (*functionName*, prefix, (*x1*.TYPE, *x2*.TYPE), *typeName*), where *x1*.TYPE and *x2*.TYPE are the type names in that variable's local declaration and all the types specified are members of TypeSet. The command

infix *functionName*;

changes a previously specified interface element to (*functionName*, infix, (*x1*.TYPE, *x2*.TYPE), *typeName*) causing the Formula IO machine to display *functionName* as an infix operator whenever it is printed. The equality operation, automatically declared for all types, has the following interface:

(= , infix, (*typeName*, *typeName*), *Boolean*)

InterfaceSet is the union of each individual interface set over all types in TypeSet.

---

interface[s] x_1[, ..., x_n]: *typeName*;

just as declare establishes the types of variables, interface provides the necessary characteristics of operators. All operators should be declared using the interface command before they are referenced in other ***Affirm*** commands. Each of the x_i will be an expression of the form *operatorName*(a_1, ..., a_m), where each of the a_i is a variable declared in the current type. The interface declaration states that *operatorName* is a function of *m* arguments, with types corresponding to those of the a_i. The value returned by *operatorName* will be of type *typeName*. In the case of an operator which takes no arguments, the parentheses may be omitted. It is also permissible to use *infix* notation, such as q apr x.

interface q apr x, apl(q, x): SequenceOfElemType;

infix *operatorName*$_1$ [, ..., *operatorName*$_n$];
each operator *operatorName* is declared to be an infix operator.

discard interface[s] *operatorName*$_1$ [, ..., *operatorName*$_n$];
discards the operations *operationName*$_i$ for *i* from 1 to *n*, in the current type. Each operator must be defined in the current type. Note that any references to the discarded operations are inconsistent. The system does not check for this condition. It is the user's responsibility to discard or redefine any rules or propositions referencing the newly-discarded operations.

4.2.3. Rules

The specification machine makes use of the InterfaceSet to enforce static type checking of expressions used in RuleSet. The elements of RuleSet are used as rewrite rules of the form Left → Right by various parts of ***Affirm***. The rule entries are three-tuples (rule class, left expression, right expression). The rule class entry indicates whether this rule is to be automatically applied whenever applicable or only under explicit user direction. The left expression and right expression entries are the left hand side and right hand side respectively of the proposed rewrite rule. These expressions are type-checked using the InterfaceSet. RuleSet is the union of each individual set of rules over all types in TypeSet.

The automatically applied rules are added to the set of rules by the incremental Knuth-Bendix convergence process to ensure that all previous rules and the proposed new rule maintain the Church-Rosser property of unique termination [§5.2]. Any new rules generated by this process are automatically added to the rule set of CurrentContext. The user-controlled rules pass through the type-checking procedure but not the incremental Knuth-Bendix convergence process. The automatically applied rules are entered by the axiom and rulelemma commands. The axioms reflect basic assumptions or definitions, while the rulelemmas are assumed to be provable from the axioms.

The commands

```
axiom Last(s apr i) = = i;
rulelemma Last(i apl s) = = if s = NewSequenceOfElemType
                            then i
                            else Last(s);
```

add a new axiom and a new rulelemma to the set of rules of the current type.

The user-controlled rules are entered by the define and schema commands. The define command often contains recursive rewrite rules and definitional notation that the user may invoke when necessary [§7.5.2.3]. The schema command specifies an induction schema that the user may later employ [§7.4.2.3]. The commands

```
define Initial(s, k) == if s = NewSequenceOfElemType
                           then s
                           else if k = 0
                                   then NewSequenceOfElemType
                                   else First(s) apl Initial(LessFirst(s), k-1);

schema Induction(s) == cases(Prop(NewSequenceOfElemType),
                              all ss(all ii(IH(ss) imp Prop(ss apr ii))));
```

add new rules to the set of rules of the current type.[16]

The discard lhs command enables the user to discard a rule. The command

discard lhs *leftExpression*;

discards the rewrite rule whose left hand side matches *leftExpression*.

---

axiom[s] a_1[, ..., a_n];

each a_i must be a rule, lhs_i == exp_i. The rewrite rule $lhs_i \rightarrow exp_i$ is (normally) added to RuleSet. Variables appearing in exp_i must appear in lhs_i. ***Affirm*** checks all proposed axioms to see how they affect the unique termination of RuleSet. It may interactively simplify the rule, reverse it, or add new rules [§5.2].

```
axioms LessLast(q apr x) == q,
       Last(q apr x) == x;
```

rulelemma[s] a_1[, ..., a_n];

As far as the system is concerned, the rulelemma command is a synonym for the axiom command. The source of rulelemmas is intended to be different, however. Axioms are basic assumptions or definitions of the data type; rulelemmas are useful primitive properties that should be provable from the axioms.

define[s] a_1[, ..., a_n];

each a_i is a rule *lhs_i* == *exp_i*. Definitions are rewrite rules, but these rules are only applied when specifically invoked by the user with the invoke command [§7.5.2.3]. Definitions are generally used to simplify notation: they are only invoked when needed, so that their contents do not overly complicate propositions. Variables in *exp_i* must either be bound quantifiers or must appear in *lhs_i, but not both.*

schema[s] a_1[, ..., a_n];

each a_i is a rule *lhs_i* == *exp_i*. The soundness of schemas is not determined by ***Affirm***; the user must establish this property. It is in schema declarations that the restriction imposed on rules is most often felt. The following declaration illustrates a very common error:

```
schema Induction(q) ==
           cases(Prop(NewSequenceOfElemType),
                 all q, x(IH(q) imp Prop(q apr x)));
```

Here the parameter *q* is the same identifier as the quantifier in the expression. A correct

[16]These examples were taken from type *SequenceOfElemType* in the PVLIBRARY.

schema declaration would be:

```
        schema Induction(q) = =
                    cases(Prop(NewSequenceOfElemType),
                          all q0, x(IH(q0) imp Prop(q0 apr x)));
```

discard lhs *lhs*;
lhs must be the left hand side of some axiom, rulelemma, definition, or schema. The rule in RuleSet with left hand side identical to *lhs* is removed from RuleSet. (This may destroy the unique termination of RuleSet; no check for this condition is performed.)

4.3. Scope Model

The ***Affirm*** system keeps track of the order in which a user opens and closes type specifications with a ContextStack. The elements of the ContextStack are types. The type and edit commands push an element onto the ContextStack; the end command closes the current type specification, and pops the top element from the ContextStack. CurrentContext, the top element of the ContextStack, is the type being specified. During specification and proof attempts, the local declarations of the CurrentContext, all interfaces of all types, and all rules in RuleSet are available to the user. Hence, the ContextStack need not contain all types used during a specification, but only the type currently being specified or used. Initially the ContextStack contains the element Basis, a type in which only the equality interface has been declared.

Affirm will stop and ask the user if it cannot determine the data type to which any given function belongs. For example, if the user types

```
type foo;
interface Null: foo;
end;

type fum;
interface Null: fum;
```

then upon each reference to the interface *Null* the system will ask the user to clarify the ambiguity.

4.4. Sufficient Completeness

How does one write a specification of a data type, and, furthermore, how can one check that the specification is, in some sense, completely specified? One idea of completeness of data types is embodied in *sufficient completeness*, so named to distinguish it from notions of completeness in logic, i.e., that every well-formed formula or its negation is provable.

A data type can be viewed as a heterogeneous algebra [V, F] where V is the set of types, v_i, and F is the set of functions, called operators, f_i [Guttag 75]. For an abstract type [V, F], the set of axioms, or axiomatization, A is sufficiently complete if, for every word of the form $f_i(x_1, ..., x_n)$, there is a theorem $f_i(x_1, ..., x_n) = u$ derivable from A where $u \in v_j$ and $v_j \in V$. Sufficient completeness is undecidable. However, there is a set of conditions, sufficient to guarantee sufficient completeness, which constitutes a semi-decision procedure for *recognizable sufficient completeness*. Intuitively, sufficient completeness is a condition which, when satisfied, indicates that the axiomatization captures the meanings of all the operators of the type being defined. These conditions, developed by Guttag [Guttag 75, Guttag 78c], and described below.

Before proceeding with the algorithm for sufficient completeness, some notation needs to be developed, and some terms need to be defined.

- The data type being defined by the axiomatization is called the *type of interest* or *TOI*.
- The set of operators F can be partitioned into two subsets called S and O. S is the set of operators whose range is the type of interest. O is the set of operators whose range is other than the type of interest. O is called the set of *output* (or *selector*) functions. Furthermore, the set S can be partitioned into two subsets called C and E. C is the set of *constructor* functions and E is the set of *extender* (or *modifier*) functions. (Several such partitionings of S may be possible.)
    * C consists of the operators which produce new values of the TOI. These constructors have the property that all instances of the data type can be represented using only operators in C. For the type SetOfElemType, for example, each set that has $n \geq 1$ elements can be represented by Insert(... Insert(EmptySet, i_1), ..., i_n) and each set that has $n = 0$ elements by EmptySet. This representation only in terms of members of C is called the *normal form*.
    * E consists of the operators which do not produce new values of the TOI. These operators are not necessary in representing values of the type of interest.
- NEST(x) is the greatest depth to which operators contained in F are nested in the expression x.
- Each axiom is of the form $f(s(x^*), y^*) = z$, also referred to as lhs = rhs, where x^* and y^* are lists, possibly empty, of free variables.
- An axiom is *conditional* if its rhs has the form *if* b *then* z_1 *else* z_2.
- A function f is *convertible* if its range is TOI and, given an axiomatization A, for all assignments to the free variables x^*, there is a theorem $f(x^*) = z$ derivable from A where z contains no occurrences of f.

The following is an algorithm for constructing a recognizably sufficiently complete data type axiomatization, as presented by Guttag [Guttag 75] and Guttag and Horning [Guttag 78c].

1. Partition F into the three disjoint subsets C, E and O.

2. Build the set CTERMS = $\{f(x_1, ..., x_n) \mid f \in C$ and $x_1, ..., x_n$ are free variables$\}$.

3. Build the set OTERMS = $\{f(x_1, ..., x_n) \mid f \in O$ and $\forall x_i$ (x_i is a free variable if $x_i \notin$ TOI; otherwise $x_i \in$ CTERMS)$\}$.

4. Build the set ETERMS = $\{f(x_1, ..., x_n) \mid f \in E$ and $\forall x_i$ (x_i is a free variable if $x_i \notin$ TOI; otherwise $x_i \in$ CTERMS)$\}$.

5. Construct a set of axioms using the members of OTERMS as left hand sides such that the resulting axiomatization of the type [V, F-E] is sufficiently complete. To do so, right hand sides should be built such that the resulting axioms satisfy the following conditions.

   - If the axiom is not conditional, NEST(lhs) > NEST(rhs).

   - If the axiom is conditional then each of the theorems lhs = z_1 and lhs = z_2 must satisfy these two conditions and either NEST(b) < NEST(lhs) or the range of f is *Boolean* and the theorem lhs = b satisfies these two conditions.

6. Construct a set of axioms using the members of ETERMS as left hand sides such that the convertibility of each member of E is shown. To do so, right hand sides should be built such that the resulting axioms satisfy the following conditions.

   - If the axiom is not conditional then there are no occurrences of f in rhs or, if f does occur, then its first parameter is a free variable contained in x* and its other parameters are free variables contained in y*.

   - If the axiom is conditional then each of the theorems lhs = z_1 and lhs = z_2 must satisfy these two conditions and either b must contain no operators in F or b must satisfy the conditions in step 5.

From the algorithm for constructing recognizably sufficiently complete axiomatizations an algorithm for discerning recognizably sufficiently complete axiomatizations, a *sufficient completeness checker*, was derived. The checker is invoked with the following command:

sufficient? *typeName*;

The sufficient completeness checker determines from the axioms the constructors, extensions and output functions and displays each set. It then examines the left hand sides of the axioms in conjunction with the sets C, E and O to determine if exactly the correct left hand sides are present. This corresponds to steps 3-4 of the algorithm. If any of the required axioms is missing, the user is informed of the situation and the checker terminates. At this point, each axiom is analyzed with respect to the conditions either in step 5 or step 6, as is appropriate. If each axiom satisfies the criteria, the axiomatization is sufficiently complete, otherwise the checker cannot determine whether it is sufficiently complete. In either case, the user is informed of the outcome.

The sufficient completeness checker is actually based on an extended version of the algorithm described above. The extensions are presented separately to avoid a more difficult presentation of the algorithm. The extensions are:

- The parameters of the type of interest need not occur first in the parameter list. Thus, for the type SequenceOfElemType in the Type Library, there is the operation

apl: ElemType $\times$ SequenceOfElemType $\rightarrow$ SequenceOfElemType.

An operation may have more than one parameter of the type of interest. Thus, for the type SequenceOfElemType there is the operation

join: SequenceOfElemType $\times$ SequenceOfElemType
$\rightarrow$ SequenceOfElemType.

- Output functions may appear in axioms that have left hand sides different from those in OTERMS. In steps 3 and 5 of the algorithm stated above for constructing data type axiomatizations, each left hand side of an axiom involving an output function, f, must be of the form f(s(x*), y*) where s$\in$C, the set of constructors. Stated simply, the set consisting of each output function composed with each constructor is exactly the set of left hand sides that must be present (for output functions) to satisfy the algorithm. However, this can be relaxed. For a given output function f, all axioms having left hand sides f(s(x*), y*), s$\in$C can be replaced with one axiom with left hand side f(x_1, ..., x_n) such that $\forall x_i$ (x_i is a free variable). For example, for the type SetOfElemType instead of the two axioms with left hand sides

IsEmpty(NewSetOfElemType)

and

IsEmpty(add(s, x))

the following axiom suffices:

IsEmpty(s) = = (s = NewSetOfElemType)

---

sufficient? *typeName*;
typeName must be a member of TypeSet. A sufficient completeness check is performed.

4.5. Type Management

4.5.1. Displaying, Saving, and Restoring Types

Affirm provides the user with commands for displaying, saving and restoring types. The command

print type *typeName* [*OptionList*];

displays information stored in the type *typeName* depending on the *OptionList* selected. The available options are declare, interface, axiom, rulelemma, defn, and schema. Any number of the options may be requested with each print. The default when no *OptionList* is supplied, is to provide all:

print type *typeName* declare, interface, axiom, rulelemma, defn, schema;

All the information stored in a type may be saved in a file by the save command [§9.6]. At a later time the type may be loaded into ***Affirm*** by the load command [§9.6], restoring all the information previously saved. The file name of the type is the uppercase version of the type name. Types can also be compiled [§9.6], since the internal representation of a data type specification is simply Interlisp code. Compiled forms of data types are more efficient to use in most cases than other forms. The exception is when the data type is still undergoing development; then the saved version is better.

---

print type *typeName*;

typeName must be a member of TypeSet. The declarations, interfaces, infix operators, axioms, definitions, and schemas of type *typeName* are printed on the terminal. Should only a subset of these be desired, *typeName* may be followed with a list of *options*.

```
print type ElemType;
print type SequenceOfElemType decl schema;
```

4.5.2. The Type Library

The ***PVLIBRARY*** contains a number of data type specifications that may be of use to the user, including sets, sequences, circles, binary trees, and of course, stacks and queues. The commands that actually read these specifications into ***Affirm*** are read, load, and, more generally, needs. These are all documented in Chapter 9. The data type specifications themselves are contained in the TYPE LIBRARY, Volume III of the ***Affirm*** Reference Library.

5. The Rewrite Rule Machine

5.1. Introduction

Affirm encourages the use of *equational specifications* for data abstractions, since the theorem prover is oriented toward performing deductions by making equational substitutions. The theorem prover is able to make such deductions automatically by treating equations, whenever possible, as *rewrite rules*. These are rules of the form

left → right

where *left* and *right* are expressions (possibly containing variables). The rules are used to rewrite expressions by replacing with *right* all subexpressions matched by *left*. Rewrite rules are applied to an expression until no further rewriting is possible; their order of application is immaterial. For example, suppose we have the rules

1. Length(NewSequence) → 0
2. Length(s apr i) → Length(s) + 1

which define how to compute the length of a sequence. *NewSequence* is the empty sequence; *apr*, append right, builds a longer sequence from two parameters, a sequence *s* and an element *i*. Given the expression

Length((NewSequence apr i) apr j)

simplification would occur as follows:

Length((NewSequence apr i) apr j)	
Length(NewSequence apr i) + 1	by rule 2
Length(NewSequence) + 1 + 1	by rule 2
0 + 1 + 1	by rule 1
2	arithmetic

An essential property of a set of rewrite rules is *finite termination*: no infinite sequence of rewrites is possible. Another extremely useful property is *unique termination*: any two terminating sequences of rewrites starting from the same expression will have the same final expression (no matter what choice is made of which subexpression to rewrite or which rule to apply first). A set of rules with the finite and unique termination properties is said to be *convergent*. If a set of axiomatic equations can be treated as rewrite rules, and these rules (or a finite set of rules derived from them) are convergent, then one can decide when an equation is provable from the axioms just by rewriting both sides to their final expressions and checking these for identity. Using rewrite rules to prove equational properties is generally much more efficient than other techniques requiring heuristic searching. Thus, the ***Affirm*** system attempts to form the equational parts of data type specifications into rewrite rules with this convergence property.

At present, the system does not attempt to prove that the finite termination property is maintained when a new rule is added to its data base, but rather assumes this property (although some simple tests are applied that may reveal its absence, in which case the rule is not added).

5.2. The Knuth-Bendix Convergence Test

The system checks for unique termination using an algorithm based on the Knuth-Bendix method [Knuth 70, Huet 78], during the processing of the axiom, rulelemma, and complete commands. The algorithm is able to generate additional rules that may restore unique termination when an added rule violates this property [Musser 80]. (Rules found to be redundant will be discarded.) The unique termination check is one of the places where the system may stop and *ask the user* a question [§3.2]. If a new rule must be generated to preserve unique termination, and the rule's direction is open to question, the user will be asked to decide.[17] If the convergence process finds a contradiction, it discards any rules it has added (including the rule the user was trying to add) and restores any rules it has discarded.

As an illustration of the Knuth-Bendix convergence process, consider the type Group:

```
type Group;
declare x, y, z: Group;
interfaces
     e, Inv(x), op(x, y): Group;
infix  op;
axioms
1.  e op x = = x,
2.  Inv(x) op x = = e,
3.  (x op y) op z = = x op (y op z);
end {Group};
```

As each rule is entered, ***Affirm*** determines any interactions with existing rules. It seeks to unify the left-hand side of one rule with some non-variable subexpression of the left-hand side of another rule. For example, rules (1) and (3) can be unified to

(e op y) op z

Since both rules are now applicable, we ask whether they ultimately give us the same result. To do this, we form a pair of expressions, according to which rule we apply first (we call this a *critical pair*). We then full simplify the pair. In the above example, the pair is

<y op z, e op (y op z)>

which simplifies to

<y op z, y op z>

If the two halves of the critical pair are identical, then the rewriting behavior of the two original rules conforms to the requirement that order of rule application be immaterial. Otherwise, we need to generate a new rule. For example, rules 2 and 3 overlap to give us

[17]The profile entry *CautiousCompletion* [§3.13] can be set to On to make the system ask about all attempted additions.

(inv(y) op y) op z

which generates the critical pair (simplified by rule 1):

⟨z, inv(y) op (y op z)⟩

We need to add this as a rewrite rule; since y appears freely on only one side, only one direction is appropriate. ***Affirm*** generates the rule

inv(y) op (y op z) → z

and proceeds.

5.3. User Interaction

When a rewrite rule is about to be added, ***Affirm*** checks it for validity and direction. Rules are subject to the following constraints:

1. All variables occuring freely on the right-hand side must be present on the left-hand side.

2. The rule must never self-loop.[18]

If a rule does not meet these criteria, or if it was generated by the system and must be confirmed, the user is asked what to do.

5.3.1. Common Responses

? Lists the available options.

Reverse
... direction. Treat rule as RHS → LHS.

Treat ... as equation → true. The rule LHS = RHS → true is used, instead.

Yes Accepts the system's choice.

5.3.2. Other Responses (Not Recommended)

Accept
The system accepts another rewrite rule (or rules) trom the terminal, processes these rules, and then processes the rule that caused this interaction in the first place.

Keep ... as is. In spite of ***Affirm*** 's objections, the rule is forced into the system.

Suppress
Discards the rule, recording it on the list BadEquations. The set of rewrite rules may no longer have the unique termination property. (No check is made of BadEquations.)

[18] A rewrite rule is forbidden if a nonvariable subexpression of the right hand side is left-unifiable with the left hand side. Expression α is left-unifiable with β if there are substitutions ρ, θ such that $\rho(\theta(\alpha)) = \theta(\beta)$. For example, unification corresponds to the case where ρ = Identity, and pattern matching to θ = Identity.

Only some of these responses are available from any one question. The particular set of choices can always be displayed by typing ?. The Users Guide contains advice on how to answer this question.

5.4. Rewrite Rule Command

The complete command invokes the Knuth-Bendix process directly.

---

complete;

Attempts to prove the Current Proposition by *reductio ad absurdum* (proof by contradiction). It does this by negating the conclusion of Current Proposition, forming a rewrite rule from it, and (temporarily) adding it to RuleSet. Each hypothesis of Current Proposition is also turned into a rewrite rule and (temporarily) added to RuleSet. The algorithm then tries to generate a contradiction in RuleSet, by performing the unique termination test. If the rule

true → false

is generated, the Current Proposition is proved by contradiction. Otherwise, the final set of rules is used to construct a new result, which may be somewhat simpler than the Current Proposition.

---

6. The Logic Machine

6.1. Introduction

The Logic Machine provides the basic underlying propositional calculus operations, as well as skolemization, normalization, unification, instantiation, and case analysis. It is used by the Theorem Prover and uses the Rewrite Rule and Specification Machines.

6.2. Propositions

Propositions are simply Boolean-valued expressions and have the form:

<u>all</u> $x_1, ..., x_n$ <u>some</u> $y_1, ..., y_m$: $(P(x_1, ..., x_n, y_1, ..., y_m))$

The keywords <u>all</u> and <u>some</u> are the standard universal and existential quantifiers of logic.

6.2.1. If-Then-Else

In ***Affirm***, all logical connectives are translated into an internal If-Then-Else form. The simplification rules associated with this form are sufficient to recognize any ground-state tautology.

The conventional operators are translated from external to internal form as follows:

not x	→	if x then <u>false</u> else <u>true</u>
x or y	→	if x then <u>true</u> else y
x and y	→	if x then y else <u>false</u>
x imp y	→	if x then y else <u>true</u>
x eqv y	→	if y then x else ~x

For example, the proposition

(P and (A imp B)) imp C

would translate to

```
if (if P then (if A then B
                 else true)
       else false)
  then C
  else true                                              (1)
```

(This will then be simplified [§6.3.1].)

6.2.2. Skolemization and Quantification

The system retains only Skolemized propositions in prenex form, those with all quantifiers in front and functions substituted for the quantified variables in the formula. The <u>all</u> and <u>some</u> lists displayed with propositions by the theorem prover are lists of Skolem functions. The Skolem functions arise from transforming a propositions in the prenex form

$Q_1x_1Q_2x_2 ... Q_kx_kF$

where for $1 \leq i \leq k$ the Q_i are quantifiers, the x_i are variables, and F is a quantifier-free logical formula, into an equivalent quantifier-free formula. Each universal quantifier and its associated variable v are deleted, and all occurrences of v in F are replaced by

$v(y_1, ..., y_j)$

where $y_1, ..., y_j$ are the existentially quantified variables preceeding v in the quantifier prefix. After all universal quantifiers have been deleted, the existential quantifiers are simply dropped. Those familiar with the resolution method of theorem proving will notice that this is the dual of the Skolemization method used there [Robinson 65]. The following example illustrates the Skolemized proposition form.

all x (some y (all z (P(x, y, z)))) $\forall x \, \exists y , \forall z(y) \; P(x,y,z))$

Instantiations of existential quantifiers by the put command [§6.5] must satisfy the Skolem dependencies of the proposition. In the previous proposition y is permitted to depend on x; thus, instantiations such as y = x or y = 0 (assuming y is an integer) are permitted. However, since z depends on y, the instantiation y = z is forbidden.

6.3. Simplification and Normalization

These operations reduce all propositions to a standard form.

6.3.1. Simplification Rules for If-Then-Else

The internal If-Then-Else form is automatically simplified in a manner similar to that of [McCarthy 63]. Consider a ternary function σ(expr, assumed, denied) that simplifies expr in the context of assumed and denied (which are disjoint sets of predicates). Our procedure can be described by the following five rewrite rules.

1. σ(if x then y else z, A, D) → σ(y, A, D), ***if x ∈ A***

2. σ(if x then y else z, A, D) → σ(z, A, D), ***if x ∈ D***

3. σ(if x then y else z, A, D) → σ(y, A, D), ***if y and z are identical***

4. σ(if (if p then q else r) then y else z, A, D)
   → σ(if p then (if q then y else z) else (if r then y else z), A, D)

5. σ(if x then y else z, A, D) → if x then σ(y, A∪{x}, D) else σ(z, A, D∪{x})
   if rules 1-4 do not apply.

The simplifier also recognizes equality operators, which it treats specially. In rule 5, if the predicate x is of the form a = b, then b = a is also added to the assumed and denied sets, providing for the commutativity of equality. Our earlier example, equation 1 [§6.2.1], would thus simplify to

```
if P
  then (if A
          then (if B
                  then (if C
                          then true
                          else false)
                  else true)
          else (if C
                  then true
                  else false))
  else true
```

6.3.2. Normalization

The function Normalize: expressions $\rightarrow$ expressions can be defined as

Normalize(x) = σ(Rewrite(x), {true}, {false})

Rewrite(x) applies all rules in the Rewrite Rule machine, and σ simplifies the conditionals.

---

normalize;

Causes the Current Proposition to be (again) normalized and printed. Since propositions are normalized upon becoming current, this will normally have no effect, but may be necessary due to the occasional incompleteness of the simplification process.

6.3.3. Case Distribution

The case analysis rules are schema: let g be a function symbol, and x_1, ..., x_n, y, z be expressions. The schematic rewrite rule

g(x_1, ..., (if x_j then y else z), ..., x_n) $\rightarrow$
  if x_j then g(x_1, ..., y, ..., x_n) else g(x_1, ..., z, ..., x_n)

raises embedded If-Then-Else expressions across function symbols to the outer level of expressions.

Such embedded conditionals arise from automatic application of rewrite rules or from (user) controlled invocation of definitions. The case analysis rules are not applied automatically.[19] Immediately after normalization, a message is printed to the alert the user to the possibility of applying the rules. However, further logical steps, e.g. replace or lemma application, may be applied to reduce the branchiness of embedded conditional expressions. Case analysis may not be applied selectively to subexpressions, hence there is potential for exponential case explosion. However the interaction between branches via simplification usually considerably reduces the final number of

[19]The profile entry *AutoCases* [§3.13] will cause automatic application of the cases command.

cases;
distributes functions over If-Then-Else's in the Current Proposition.

Do not confuse this command (distributing conditional expressions) with the cases expression (expressing proof division in schemas) or with the suppose command that bifurcates the proof tree.

6.4. Evaluation

eval *expression*;
Simplifies *expression*, and prints the result. This is useful for testing and demonstrating abstract data types.[20] For more details on its use, see the Users Guide.

set *variable* to *expression*;
variable no longer represents itself; it is assigned a value (which will replace it whenever an expression is normalized]. (This effect is permanent until *variable* is explicitly given another value.) This may be useful in conjunction with the eval command. Other than that, it is not recommended.

6.5. Instantiation and Unification

These commands provide a means of assigning values to existentially-quantified variables.

put $v_1 = e_1$ [, ..., $v_n = e_n$];
Each of the v_i must be a variable in the some list of the Current Proposition. Each of the e_i is an expression upon which the corresponding v_i can legally depend [§6.2.2]. The e_i are substituted for the corresponding v_i.

let $v_1 = e_1$ [, ..., $v_n = e_n$];
Has the same effect as the put command, except that the new result is the *disjunction* of the unchanged and the instantiated versions of Current Proposition. Thus, all variables in the some list remain subject to further instantiation with the put or let commands. This is useful if the user is not quite sure about an instantiation, or wishes to perform multiple instantiations. (It does, however, double the size of the expression.) If, for example, the Current Proposition was

```
all x (some y (P(x, y)))
```

The command

```
put y = x;
```

[20]The profile entry *ShowRules* [§3.13] is useful for observing the application of axioms to sample expressions.

would give

    all x (P(x, x))

while the command

    let y = x;

would yield

    all x (some y (P(x, y) or P(x, x)))

6.6. Chaining and Narrowing

The search command implements a procedure called *chaining and narrowing* [Lankford 78]. Chaining can be viewed as a generalization of Simplification Rule 4 that propagates assumptions and denials through branches of conditional expressions. The generalization occurs in using the *most general unifier* of subexpressions in the condition and branch expressions. For example,

    if P(x) then P(a) else true

can be chained to produce

    if P(a) then true else true

Narrowing is the application of the usual simplification rules to the formulae resulting from chaining. So the chaining and narrowing procedure is nothing more than determining unifiers, applying them, and normalizing. The effect of the algorithm is the determination of whether a quantifier-free first-order sentence (arising from Skolemization) in If-Then-Else form has a ground instance.

In practice, this means that the algorithm simply tries all instantiations, normalizing each instantiated formula, and stopping if an instantiation results in reducing the proposition to true. The output of the algorithm may be any of the following:

1. Unsuccessful: no unifications can be found.

2. A list of labeled instantiations culminating in unsuccessful.

3. The above list of labeled instantiations followed by the effect of an automatically applied put of the effective instantiation, giving the result true.

Search essentially consists of a possibly large number of evaluations, with the associated cost.

In the second case above, the auxiliary command choose permits selection of an instantiation by its hierarchical labels, in effect a put of that instantiation. The list of instantiations is not maintained but instead is re-generated up to the point of the specified choice. Hence, choose causes little additional space consumption.

Search should be applied either where its success is not expected, but its instantiation list is desired, or where its success is fully expected in that an instantiation is probable, but perhaps lengthy or tricky to enter. Search is not meant to be used in the connotation "go to work, algorithm, and see what you can do for me."

---

search;

Uses the method of *chaining and narrowing* to attempt to automatically find the instantiations sufficient to reduce Current Proposition to true. The command displays the sets of instantiations it tries. These may be referenced by the user in the choose command.

choose *path*;

Related to the search command, this command allows the user to pick some sequence of instantiations tried by the search command. The search command prints a small integer label to the left of each instantiation it attempts. The sequence of numbers describing the choice--*path*--is the parameter to the choose command. This command is useful if search found lengthy instantiations, but was unable to achieve a final proof.

6.7. Integer Simplification

Simplification of integer expressions contained in propositions occurs both automatically and at explicit user request.

The automatic simplification is part of the normalization process. For a very few integer operations, the system essentially adds more information to the sets of Assumed and Denied propositions at each application of the five If-Then-Else evaluation rewrite rules [§6.3.1]. For example, suppose the current proposition under proof attempt, P, is

```
if i < j
  then x
  else y
```

Then $\sigma(P, A, D)$ =

$$\begin{aligned} & \sigma(\text{if } i<j \text{ then } x \text{ else } y, A, D) \\ = \; & \text{if } i<j \\ & \quad \text{then } \sigma(x, A \cup \{i<j\} \cup \{i\leq j\}, D \cup \{j\leq i, j<i, i=j\}) \\ & \quad \text{else } \sigma(y, A \cup \{j\leq i\}, D \cup \{i<j\}) \end{aligned}$$

This incorporates the knowledge that $i<j$ implies $i\leq j$ and $j\not<i$, and that $i\not<j$ implies $j\leq i$.

Extra information is only added for the integer inequality and equality relations, as follows.

Operation	Extra Information			
	Then-Branch		Else-Branch	
	Assumed	Denied	Assumed	Denied
=	i = j, i ≤ j, j ≤ i	i < j, j < i		i = j
<	i < j, i ≤ j	j ≤ i, j < i, i = j	j ≤ i	i < j
≤	i ≤ j	j < i	j < i, j ≤ i	i = j, j ≤ j, i < j

The system adds these extra facts only temporarily, during the normalization process. These facts are not added permanently to the proposition. If the user needs to introduce a hypothesis that is deducible from the hypotheses already present in the current proposition, the best approach is to use the suppose command [§7.4.2.2], which will split the proposition up into two (simpler) propositions. One of these two should be trivially true.

More explicit integer simplification is provided by the normint command (normalize integers). This command in an implementation of *separation theory* [Pratt 78], a theory complete over integer constant addition. Separation theory basically keeps track of the minimum non-negative separation between each pair of integer terms. A non-zero separation means one term is less than the other; a zero separation means the two terms are equal. A directed graph structure is used to build up the known separations between each pair of terms in the current proposition. The transitive closure of the graph with respect to this separates relation thus represents the complete known relationships among the terms in the proposition. A cycle with non-zero separations indicates a contradiction, as is displayed in the following example. Suppose the current proposition is

```
     i < j
and j < k
and k < i
imp C
```

The transitive property of the "<" operator and the first two hypotheses imply that i < k, which is directly contradicted by the third hypothesis.

The directed graph structure representing the known separations between integer terms is built up from the individual integer inequalities as follows.

i < j (or i + 1 ≤ j)

$$\begin{tikzcd} /i/ \arrow[r, "1"] & /j/ \end{tikzcd}$$

j < k (or j + 1 ≤ k)

$$\begin{tikzcd} /i/ \arrow[rr, "1"] & & /j/ \arrow[dl, "1"] \\ & /k/ & \end{tikzcd}$$

k < i (or k + 1 ≤ i)

$$\begin{tikzcd} /i/ \arrow[rr, "1"] & & /j/ \arrow[dl, "1"] \\ & /k/ \arrow[ul, "1"] & \end{tikzcd}$$

At this point, the graph contains a cycle with non-zero minimal separations, indicating a contradiction. The hypothesis is ~~there~~ false, making the CurrentProposition true.
therefore

---

normint;
: invokes the algorithm employing *separation theory* to simplify the current proposition using the integer inequalities contained in it.

6.8. Output

6.8.1. Translation of Internal Form to External Form

Complicated logical expressions can take many interchangeable forms. Once the system has converted propositions into the internal If-Then-Else form, it has no way of recalling how they should be printed. (For example, does the user want to see "A imp B" or "not(A) or B" ?) It uses a series of heuristic rewrite rules to produce normal logical connectives from the internal form. Generally, the system deals well with implications and conjunctions, but not quite as well with equivalences and disjunctions. The following rewrite rules summarize the transformations from internal to external form:

```
if B          → X
  then X
  else X

if true       → X
  then X
  else Y

if false      → Y
  then X
  else Y
```

```
if B                     → B
  then true
  else false

if B                     → ~B
  then false
  else true

if B                     → B or Y
  then true
  else Y

if B                     → (~B) and Y
  then false
  else Y

if B                     → B and X
  then X
  else false

if B                     → B imp X
  then X
  else true

if B                     → X eqv B
  then X
  else ~X

if B1                    → if (B1 and B2)
  then (if B2                then X
            then X           else Y
            else Y)
  else Y

if B1                    → if (B1 or B2)
  then X                     then X
  else (if B2                else Y
           then X
           else Y)

if B1                    → if (B1 and (~B2))
  then (if B2                then Y
            then X           else X
            else Y)
  else X

if B1                    → if ((~B1) and B2)
  then X                     then Y
  else (if B2                else X
           then Y
           else X)
```

6.8.2. Printing Variant Forms of Propositions

print result;
prints the Current Proposition [§7.2] in its normalized form.

print variables;
lists just the *variables* in the Current Proposition.

7. The Theorem Prover Machine

7.1. Introduction

Almost all ***Affirm*** users will have theorems to prove. These may simply be useful lemmas about the data types, stated and proven in order to verify that the specification matches its intuitive counterpart. They may be significant theorems whose proof is the ultimate goal of the ***Affirm*** session. Or they might be verification conditions for a program.

Whatever the origin of these propositions, they will be subjected to the theorem prover. Some of its important characteristics are:

- It automatically simplifies expressions by applying rewrite rules from the axiomatically-specified data types [Chapter 5].
- The proof process is user-directed -- one effectively "walks the system through a proof". The user makes all strategic decisions; the system carries them out and displays the results.
- A natural-deduction style is followed, in the sense that one sets up goals and repeatedly splits each of them into (perhaps several) simpler subgoals. The objective is to generate a set of terminal subgoals, all of which are directly deducible from the axioms. The state of a proof-in-progress is represented as a <u>Proof Tree</u>.
- Subgoals are proved *independently*.[21] An instantiation in one branch is completely independent of any instantiations done in parallel branches. When soundness does not permit such independent proving, ***Affirm*** forbids the split.
- Incomplete steps and unproven lemmas are monitored in order to assure the integrity of the proof process. Circular reasoning is not allowed.

The User's Guide contains "An Introductory session with ***Affirm***"; the session introduces many of the theorem prover commands. In addition, Appendix VIII contains a very brief synopsis of a subset of the possible system commands.

7.2. Key Data Structures in the Theorem Prover

Propositions are Boolean expressions to be proven [§6.2]. The system has a cursor, which always rests on one of them (designated the <u>Current Proposition</u>).[22] This is the one upon which theorem proving commands act, and is one's goal, out of which we seek to generate subgoals.

[21]Of course, lemmas may be shared among different parts of a proof.

[22]Initially, the <u>Current Proposition</u> is the constant <u>true</u>, which cannot be further proven.

Certain propositions are included in the set Theorems;[23] one's primary task is to prove them, although in the process they will probably be broken into easier subgoals. Verification conditions [§8.3] are theorems, as are conjectures entered by the user.

Since proofs proceed by the generation of subgoals, a proof can be viewed as a tree (rooted in the theorem). Nodes correspond to propositions, and arcs record the subgoaling relationships between them. Associated with each node is the ***Affirm*** command (if any) by which its demonstration has been attempted. Consequently, the state of the theorem prover can be summarized by the Proof Forest, a collection of such trees. The theorem (if any) whose proof tree includes the Current Proposition is called the Current Theorem.

Since lemmas are often used to partition a proof in order to reduce complexity, the proof tree of a lemma is kept separate from those of any theorems which apply it. Instead, the logical dependency established by lemma application is recorded in a separate graph, the *uses* relation. (Remember that lemmas, since they are user-entered conjectures, are included in the set of Theorems.)

In order for a theorem to be proven, it must use only proven lemmas, and all leaves in its proof tree must be reducible to true. ***Affirm*** monitors both of these conditions in order to notify the user as progress is made and to provide help in identifying and selecting unfinished parts of a proof. We use the term unfinished leaves to refer to the non-true propositions on a tree's frontier. Each theorem has a *proof status*, which is one of the following:

- *untried*: no proof attempted
- *tried*: has a proof tree, but still has unfinished leaves
- *awaiting lemma proof*: proof tree is completed, but some of its lemmas are unproven
- *proven*: tree complete, all lemmas proven or assumed
- *assumed*: has been so designated by the assume command [§7.9.1].

Affirm announces as a theorem progresses from state to state; for example, it might say

```
theorem Main awaiting proof of lemmas SeqLarger and SeqFact.
```

Propositions may be named [§7.9.2], or annotated with an arbitrary comment [§7.9.3]. If the user does not supply a name when stating a lemma, one is automatically generated. Unnamed positions within a tree are assigned a unique *node number* so the user can still refer to them. If a and generates more than one subgoal, each is identified by an arc label. For example, the

nomenclature: we use the term theorem to refer to conjectures, whether proven or not. This is in the spirit n; the user's job is to demonstrate that all the conjectures in the set Theorems are *really* theorems. Thus orems and unproven theorems. Sometimes, the unproven theorems are in fact found to be false and cases of an induction on sequences might be labeled with *NewSequence:* and *apr:*.[24]

7.3. Theorem Creation

Theorems are entered into ***Affirm*** by the commands theorem, try [§7.6.1.1], use [§7.4.1.2], apply [§7.4.1.1], and genvcs [§8.3].

theorem [*nodeName*,] *proposition*;

This command simply enters the proposition into Theorems, and creates a root in the Proof Forest that may later be attempted. It does not affect the Current Proposition. The user may associate a name with the theorem. This command is especially useful for command files containing lists of theorems.

7.3.1. A Note on Syntax

Several ***Affirm*** commands allow a target to be specified using the syntax

[*nodeName*,] proposition

This permits one to refer to a known proposition by name or expression, or to enter a new one. At the same time, a name may be assigned, overwriting any previous one for this expression. Here are some examples.

apply Easy1;	*{Easy1 is already known to **Affirm**}*
apply EasyVc, SortUpwards # 3;	*{rename a known propn while applying it}*
theorem PandR, P(x) imp R(x, f(x));	*{name something new}*
try P(f(x)) imp P(x);	*{In case user remembers expression but not name. Uncommon.}*

7.4. Proof by Several Subgoals

A crucial element of a proof strategy involves deciding how to divide a proof into subgoals which can then be attacked independently. Properly done, this can be the key to a manageable proof for a complex theorem.

7.4.1. Lemma Application

The user will often make use of lemmas. These are theorems in their own right: meaningful statements about the data types, proven separately from the Current Theorem.

7.4.1.1. apply [*nodeName*,] *proposition*;

This command places *proposition* in Theorems, and adds it as a hypothesis to the Current Proposition. The command records this dependency by establishing the Uses relationship between the Current Proposition and *proposition*. The expression corresponding to

[24] By convention all arc labels end in a colon, so they can be distinguished from node names.

proposition will have its variables renamed to avoid conflicts; the renamed form is printed on the terminal. The resultant Current Proposition is not printed,[25] since no meaningful simplification will occur until the user has performed instantiations. Typically, a put or search command will follow apply [§6.5].

7.4.1.2. use [*nodeName*,] *proposition*;

Like apply, but prints the new Current Proposition.

7.4.2. Case Analysis

When a proof applies a lemma, some generally meaningful fact simplifies an intermediate proof step. Alternatively, one may ask that such an intermediate expression be subjected to case analysis, yielding two or more pieces to be proven separately. (Because these proofs are independent, certain constraints are enforced concerning some variables.)

7.4.2.1. augment *proposition*;

proposition is added as a hypothesis to the Current Proposition. Separately, the user must show that *proposition* can be deduced from the hypotheses already present. Any free variables in *proposition* are identified with those in the Current Proposition, rather than being renamed. Given a Current Proposition of the form

H imp C

this command spawns the two children:

- H imp *proposition*

- (H and *proposition*) imp C

These children are assigned the arc labels thesis: and main:, respectively.

7.4.2.2. suppose *proposition*;

This command splits the Current Proposition into two (and sometimes more) cases:

- *proposition* imp Current Proposition

- *proposition* or Current Proposition

These children are labeled yes: and no:.

When *proposition* is omitted,[26] the splitting predicate is automatically generated by ***Affirm*** using the

[25]But see the use command [§7.4.1.2].

[26]The suppose command without a parameter replaces the obsolete split command.

internal If-Then-Else form of Current Proposition. Basically, the predicate is chosen from the first significant branch point. For example, if the Current Proposition is of the form

```
        A imp B
      and H
   impC
```

the suppose command will yield

```
        A
      and B
      and H
   impC
```

and

```
        (~A)
      and H
   impC
```

A detailed description follows, but *it is usually best just to experiment*. The children generated by the suppose command are labeled first:, second:, etc. Usually there are only two.

If Current Proposition is of the form:	The children are:
if B then C_1 else C_2	{B imp C_1, B or C_2}
C_1 and C_2 and ... and C_k	{C_k, C_1 and ... and C_{k-1}}
H imp (C_1 and ... and C_k)	{H imp C_1, (H and C_1) imp C_2, (H and C_1 and C_2) imp C_3, ..., (H and C_1 and ... and C_{k-1}) imp C_k}
H_1 and (H_2 imp C_1) and H_3 imp C_2	{(H_1 and H_2 and C_1 and H_3) imp C_2, H_1 and (~H_2) and H_3 imp C_2}
(H_1 or H_2) imp C_1	{H_1 imp C_1, (~H_1) and H_2 imp C_1}

7.4.2.3. employ *schema(var)*;

This directs ***Affirm*** to set up a proof using a schema. Usually, a schema is used for data-type induction or normal form arguments [§4.2.3]. See the User's Guide for a description of how schemas are defined, and what cases they produce.

Schema must have been defined for objects of *var*'s abstract data type. The various cases are set up as children of the Current Proposition; ***Affirm*** simplifies them and announces which ones are immediately proven, and which remain to be attempted manually. If there is more than one child, they are automatically given arc labels derived from the main operator of each case.

Employ can only be used when *var* is contained in the all list, and has no dependencies upon any variables in the some list [§6.2.2]. For example, *y* may not be inducted upon in the expression

```
some x (all y (P(x, y)))
```

because otherwise one could give *x* conflicting instantiations in the different cases. If this restriction were not enforced, one could incorrectly prove that

```
some x (all y (x > 1 and remainder(y, x) = 0))
```

A Detailed Look at the Processing of an employ Command

IH is the inductive hypothesis, and Prop is the proposition to be proven. These expressions are generated when the user issues the employ command, as follows. Suppose the induction schema has the following definition:

```
schema Induction(s) = = cases(Prop(Zero),
                              all ss (IH(ss) imp Prop(Succ(ss))));
```

This is of course just the induction schema for non-negative integers. If the proposition to be proved is P(i, j), for non-negative integers *i, j*, then when the user issues the command

```
employ Induction(i);
```

the following actions occur.

- IH and Prop are defined:

  ```
  axiom Prop(i) = = all j (P(i, j));

  define IH(i)  = = all j (P(i, j));
  ```

  That is, Prop is defined as an automatically-applied rewrite rule, and IH becomes a definition, to be explicitly invoked by the user.

- After the above two rules are defined, each of the cases of the induction schema is *normalized*. All of the rewrite rules in the system are used, as usual, in the simplification and normalization process. This includes the axiom above defining Prop.

  ```
  Prop(Zero)
  ```

  becomes

  ```
  all j (P(Zero, j))
  ```

  by application of the axioms defining Prop; further application of other rewrite rules may well simplify the proposition even further. The induction step

  ```
  all ss (IH(ss) imp Prop(Succ(ss)))
  ```

  becomes

  ```
  all ss (IH(ss) imp (all j (P(ss, j))))
  ```

  when the axiom defining Prop is applied; the embedded quantifier(s) will 'bubble' to the outer context during Skolemization, so that the whole proposition becomes

  ```
  all ss, j (IH(ss) imp P(ss, j))
  ```

  and of course, further simplification will probably occur as a result of the application of other rewrite rules.

- The two normalized propositions, corresponding to the two cases in the schema definition, are then added to the proof tree of the Current Theorem by making them children of the Current Proposition. Proof of both of these children is then assumed to validate the proposition trying to be proved. Notice that this assumes the schema itself is a valid inference rule. Currently ***Affirm*** makes no attempt to validate this assumption. When the user defines a schema *Ind* for a data type *DT* with cases *A*, *B*, *C*, intuitively the user is filling in *only the top line of an inference rule, where the system has already filled in the bottom line*: given the command

```
schema Ind(dt) = = cases(Prop(A), Prop(B), Prop(C));
```

the system generates an inference rule as follows, *with no further validation*:

```
Prop(A), Prop(B), Prop(C)
     all dt (Prop(dt))
```

In other words, the system always assumes the cases defined by the user cover the entire set of values of the data type. Therefore the system always generates an inference rule with a consequent of the form

```
all s (Prop(s))
```

The Usè of IH

Once the employ command has split up a proposition into a number of cases, each smaller and hopefully simpler proposition is attempted on its own. Many will be of the form

```
all ss, j (IH(ss) imp P(ss, j))
```

where P is the proposition that the user employed the induction schema upon in the first place. IH is nothing more than an unexpanded P:

```
IH(ss) = = all j (P(ss, j))
```

The main reason for the deferment of the expansion is to limit the large number of changes the original proposition undergoes when a schema is employed. The deferment allows the expansion to occur in several smaller steps, instead of one huge one. Given the above proposition form, the user would probably expand the IH reference,

```
invoke IH;
```

and the proposition being proved would become

```
all ss, j (     (all j (P(ss, j)))
        impP(ss, j))
```

which after Skolemization would be

```
all ss, j (some j' (P(ss, j') imp P(ss, j)))
```

The reader can see that it is oft-times desirable to take these steps one at a time, rather than all at once.

Another reason for deferring the expansion of IH is that a lemma may need to be applied, or a definition other than IH may need to be invoked, that further simplifies the proposition before adding the complexity of the detailed induction hypothesis.

The references to the induction hypothesis IH in schema definitions take exactly one parameter, of the type being defined. However, when the references to IH appear in actual propositions to be proven, they will have *two* parameters. The first is the usual value of the type inducted upon; the second is a *node number*, the name of the node where the employ command was issued. This is used to distinguish various IH's in different proof attempts (or multiple IH's in the same proof attempt). The system attempts to be helpful, and map this number into a user-defined name whenever possible. The user-defined name is displayed in comment brackets to signify the fact that upon input of any expression involving a two-parameter reference to IH, the second parameter must be an integer node number, rather than a user-defined theorem name.

7.5. Proposition Transformation

Obviously, one has to be able to do something with propositions other than fragment them; at some point, the pieces have to be proven. They must be stepwise transformed, in a sound way, until they reach true. Such transformations conveniently fall into two classes: instantiation, and everything else. They all produce a single child, the new result.

7.5.1. Instantiation

The commands put, let, search, and choose assign values to variables in the some list of the Current Proposition [§6.5].

7.5.2. Other Transformation Commands

The remaining commands all produce something which is logically equivalent to Current Proposition (in the context of the data type axioms). The result is presumably in a more workable form.

7.5.2.1. complete ;

Uses the Knuth-Bendix method to seek a proof by contradiction [§5.4].

7.5.2.2. replace [*expression*$_1$ [, ..., *expression*$_n$]];

replace reasons about equalities. This command can be automatically invoked by the *AutoReplace* profile entry [§3.13].

If no argument is given, then hypotheses in the Current Proposition of the form L = R are used to replace all other occurrences of L with R. If arguments {*expression*$_i$} are given, each should occur in an equality hypothesis of the form

$$expression_i = R_i$$

or

$$R_i = expression_i$$

All other occurrences of *expression*$_i$ are replaced with R_i. For example, if Current Proposition is

```
        fee(j, k)
      and j = m
      and n = k
   imp fie(m, n)
```

replace; will yield

```
        fee(m, k)
      and j = m
      and n = k
   imp fie(m, k)
```

while the command replace m, n; will yield

```
        fee(j, k)
      and j = m
      and n = k
   imp fie(j, k)
```

Unfortunately, if a goal contains several equations replace may not work the first time. For example, the proposition

```
        fee(x, i)
      and i = j
      and j = k
   imp fee(x, k)
```

would require two replaces to chain the equalities. Often, experimentation is useful; one may need to undo the replace and make it more specific and/or swap some equalities. (See the User's Guide.)

7.5.2.3. invoke *rangedOp*$_1$ [, ...];

Each of the specified operators should occur in the Current Proposition and have a definition [§ 4.2.3]. The definition is expanded. (If an operator appears in its own definition, the new occurrence will not be expanded; thus the process will not loop.) An ordinal range may be specified; if it is not, the first occurrence of each operator will be expanded. Some examples:

invoke IH;	invoke the first IH
invoke IH |1|;	"
invoke IH |2|;	second IH
invoke IH |all|;	all IH's
invoke IH |last|;	invoke the very last IH
invoke IH |-1|;	"
invoke IH |-2|;	next to last
invoke IH |2:4|;	second, third and fourth
invoke F(i,j)|2:5|, G|3,5|;	second through fifth occurrences of F(i,j) and the third and fifth occurrences of G

This command can be automatically invoked by the *AutoInvokeIH* profile entry [§3.13].

7.5.2.4. swap ***$rangedExp_i$*** [, ...];

This command reverses equality hypotheses in the Current Proposition. Thus, it is often useful in conjunction with the replace command [§7.5.2.2]. Each of the *$rangedExp_i$* specifies one or more equalities to be reversed. Such a specification may give one of the arguments to the equality, or an ordinal range, or both. For example:

swap a;	Swap all equations whose left hand side (or right hand side) is the expression *a*.
swap |2|, |-2|;	Swap the second equation, and the next-to-last equation.
swap a |-1|;	Swap the last equation whose left-hand or right-hand side is the expression *a*.

Note that "a ~= b" is really "~(a = b)".

7.5.2.5. denote ***expression*** by ***variable***;

Often the same expression will appear in several places in a proposition. If the common subexpression is large, it can make the proposition confusing to read. In some other cases, it would be useful to perform induction on the value of an expression; ***Affirm***, however, only allows this to be done for variables.

The solution to both of these problems lies in denote. It replaces all occurrences of *expression* with *variable*, and adds the hypothesis

expression = variable

to the proposition. Thus, if one wishes to expand these occurrences back out, one may issue the commands[27]

swap *variable*|1|;
replace *variable*;

Variable must not occur anywhere in *expression*. If the variable is declared, it must be of the proper type. If it is not declared, ***Affirm*** will do so automatically. The variable may also be renamed to avoid name conflicts with any other variables bound in the proposition.

7.6. Cursor Movement

These commands all change the Current Proposition, but are nondestructive; they leave the Proof Forest unchanged.

[27] Intervening commands might cause other equalities involving *variable* to precede this one, in which case the ordinal number 1 would not be appropriate for swap.

7.6.1. Absolute Movement

7.6.1.1. try [*nodeName*,] *proposition*;

Makes *proposition* be the Current Proposition. If *proposition* is in Theorems, it becomes the Current Theorem; otherwise, this designation is applied to its parent theorem. (If *proposition* is new or an orphan, then it is added to Theorems.) *proposition* is normalized and printed. This command is used for

- random access in a proof tree; and

- starting or resuming a proof (but see the description of resume [§7.6.1.2]).

7.6.1.2. resume ;

The Current Theorem must be tried. The Current Proposition is restored to the value it had when the user was last proving this theorem, thus resuming a partially-completed proof. (This command is usually preceded by a try command.)

7.6.1.3. retry ;

Retries the Current Theorem.

7.6.1.4. next ;

Moves to the next task, according to a depth-first plan, using the following hierarchy:

1. If the Current Theorem has unfinished leaves, move to the next one.

2. If the Current Theorem applies an unproven lemma, try it.

3. If the Current Theorem is applied as a lemma by an unproven theorem, return to it. (This process extends to any unproven ancestor.)

4. If none of the above hold, then stay put and perform the command

```
print status unproven;
```

Within this hierarchy, we prefer the most-recently-attempted theorem. Where possible, resume. This command can be automatically invoked by the *AutoNext* profile entry [§3.13].

7.6.2. Relative Movement

These commands all operate in terms of the Current Proposition's position in the proof tree.

7.6.2.1. up [*integer*];

Moves the cursor up to its immediate parent in the tree. If the Current Proposition is already a theorem, this command has no effect. The number of ascensions (default 1) may be provided.

7.6.2.2. down [*Child*];

The Current Proposition must have children; this command descends to one of them. *Child* may be:

- an arc label;

- the name of a child;

- an ordinal number (between 1 and the number of children of Current Proposition);

- a node number (if *Child* > # children) (*This option is not particularly recommended*); and

- omitted: the first untried child is picked. (That failing, the first child is picked.)

7.6.2.3. arc *arcLabel*;

Is used to move between cases. Somewhere above Current Proposition is a node with a child labeled *arcLabel*. That child becomes the new Current Proposition. For example, if an induction has three cases (Zero:, Plus:, and Difference:), the user might wish to proceed in an unusual order, saying

```
arc Plus;
...
arc Difference;
...
arc Zero;
```

7.7. Printing

The print command takes a host of forms. In the following, *whatNodes* may be taken to be one or more of

- * for Current Proposition

- T for Current Theorem

- `theorems` for all the members of Theorems

- `named` for all the named propositions

- a *proposition*

7.7.1. print status [*whatNodes*];

Tells whether the specified theorems are tried, untried, awaiting lemmas, proved, or assumed. The default when *whatNodes* is omitted is `theorems`.

7.7.2. print uses [*whatNodes*];

Which lemmas are applied where? The default is `theorems`.

7.7.3. print assumptions;

Lists all the assumed propositions, and which theorems depends on them.

7.7.4. print proof [list | nolist] [theorems | *whatNodes*];

Displays the proof tree of the selected theorems (the `theorems` options selects all the members of Theorems). The default is T: i.e., the Curremt Theorem. `List` causes any lemmas that are applied to the proof of Current Theorem to be also listed. (Note that the target does not have to be a theorem, so the user can print a partial proof tree.) This command can be automatically invoked when a proof is first completed by the *AutoPrintProof* profile entry [§3.13]. The `theorems` option of print proof can be automatically executed when the user types quit via the *AutoPrintProofTheorems* profile entry [§3.13].

7.7.5. print both [list | nolist] *whatNodes*;

Like print proof but lists all the propositions in the proof tree. *Verbose and rarely useful.*

7.7.6. print prop *whatNodes*;

Lists the propositions and their associated names. For example,

    print prop T;

prints Current Theorem.

7.7.7. print known nodes;

Lists the set of known proposition names.

7.7.8. print next;

This command displays the proposition that the next command would make the Current Proposition.

7.7.9. print status unproven;

Prints the status of all unproven theorems.

7.7.10. Other forms of print

print type is used to view a data-type specification [§4.5.1]. print variables, print result, print original, and print IH print various pieces of the Current Proposition [§6.8.2].

7.8. Node Sharing; Old Proof Attempts

Nodes in the Proof Forest correspond one-to-one with propositions. This means that a particular subgoal can only have one proof at a time, even if the subgoal was generated from more than one theorem.

Propositions, once generated by ***Affirm***, are never forgotten unless they are explicitly discarded [§7.10.3]. If a proof tree is changed so as to produce a different subgoal, the disconnected subtree is still remembered, for it is associated with the old subgoal and its children. If a subsequent command causes that subgoal to be regenerated, its proof subtree will automatically reappear.

For example, suppose that some sequence property P(s) is undergoing a proof attempt. The command

```
employ Induction(s);
```

sets up subgoals

```
P(New)
```

and

```
IH(s1) imp P(s1 apr i)
```

But after working on both branches, the user decides to use a different approach. Using try, up, or retry, as is appropriate, the user goes back to the point of the old employ, and issues the new command

```
employ FirstInduction(s);
```

Now the subgoals are

```
P(New)
```

and

```
IH(s1) imp P(s1 apl i)
```

Since the first subgoal is the same as before, its proof remains in the tree. The second, new subgoal will have an empty tree (unless it has also been seen before). If the employ is later changed back to its original value, both old subtrees will be reattached.

7.9. Node Modification

These commands do not move the cursor or transform any nodes, but instead attach information to a specific node.

7.9.1. assume [*nodeName*,] *proposition*;

Marks *proposition* as assumed: it is as if this node were proven (except that this special status is remembered). It may (and should) be given a name; this is useful if a file lists assumed facts (such as integer lemmas).

7.9.2. name *nodeName* [, *proposition*];

Used to *rename* nodes [§7.3.1]. Merely notices the proposition if it is not already known; does not put it into Theorems.

7.9.3. annotate [*nodeName*,] *Annotation*;

Attaches a comment to *proposition*; this will appear whenever *proposition* does. *Annotation* is arbitrary text, but cannot contain any semicolons. This is useful for

- documenting where and when an assumption was proven;
- noting what the user's plans are when the proof attempt returns to this spot; and
- commenting a tricky place in a proof.

This command can be automatically invoked by the *AutoAnnotate* profile entry [§3.13].

7.10. Proof Tree Maintenance

These commands clear away unwanted information from the theorem prover.

7.10.1. clear proof;

Empties the Proof Forest and Theorems. Erases all proposition names, annotations, and assumptions. (Fortunately, this command is undo-able.)

7.10.2. discard theorem *nodeName*$_1$ [, ..., *nodeName*$_n$];

Removes the designated nodes from Theorems. It thus no longer has a proof state, and disappears from summaries of theorems. This is useful when an incorrect lemma has been stated. It is not permissible to discard a lemma which is applied in the proof of some other theorem. (If one of the *nodeName*$_i$ applies another as a lemma, that is okay. ***Affirm*** sorts the list first, and removes the *uses* relationship when the using theorem is deleted.)

These nodes continue to exist, and retain their proofs; they form part of the disconnected nodes in the tree. The try command will reverse the effects of discard theorem.

7.10.3. discard disconnected ;

Any nodes which are disconnected (not part of the proof tree of any theorem) are destroyed. Their expressions, annotations, names, and proofs go away. This can save a considerable amount of space. Since the command is undoable, space is only reclaimed when this event is forgotten [§3.4].

8. The VC Generation Machine

8.1. The Programming Language

The programming language accepted by ***Affirm*** is an extension of a subset of axiomatically defined Pascal. The extensions include import lists as in ***Euclid***, minor extensions to statement syntax and semantics, and a more uniform treatment of expression syntax. Expressions (and in particular, functions) may not have side effects. Statements may include embedded assertions expressed in the predicate language subset of the specification language. The programmer may introduce pre and post conditions on procedures and functions, invariant assertions and/or subgoal assertions on while, for, and repeat statements, and asserting clauses on go to and return statements. The programmer may also introduce assertions at any point via the assert statement.

The full grammar for the accepted language appears in Appendix II. Here are some examples of the strictly non-Pascal constructs:

1. Expression precedence follows mathematical conventions. For unary and binary operators the hierarchy is as follows:

unary not, unary minus	(highest precedence)
user defined infix operators	
expt	
*, div, mod	
+, -	
$=, <, >, \leq, \geq, \neq$	
and	
or	
imp	
eqv	(lowest precedence)

   To illustrate, let *in* be a user-defined binary infix operator.

   a. `not i in x` means `(not i) in x` which is different from `not(i in x)`

   b. `i + j in x` means `i + (j in x)` which will probably produce an interface error. The intent is most likely `(i + j) in x`

2. Procedures and functions[28] are declared as follows:

   ```
   procedure p(var x: Integer; y: Integer)
       imports (z, w: Boolean; var v: Integer); pre ...

   function f(x: Integer; y: Boolean)
       returns w: Integer imports (z: integer); pre...
   ```

[28] As of ***Affirm*** version 1.21, the verification condition generator does not process functions correctly. The user is advised to use only procedures.

These examples show the position of the optional import list. Functions may not have var parameters, either formal or imported. The return construct for functions should also be noted. Assertions may be introduced at the beginning of a block by

pre assertion:
post assertion;

3. In actual procedure calls, there may be no aliases, which means that no two var parameters, either formal or imported, may have the same actual parameters. The reason for this restriction involves the substitution rules in use [§8.4].

4. Each statement is optional; a missing pre or post condition is taken as the constant true. Assertions may be introduced where statements may appear via

assert assertion;

5. Assertions may be added to loop statements as follows:

maintain assertion$_1$
while expression do statement
thus assertion$_2$;

repeat statement$_1$; ...; statement$_n$ until expression
thus assertion;

maintain assertion$_1$
for ... do statement
thus assertion$_2$;

6. Go to and return statements may have asserting clauses as follows:

go to label asserting assertion;
return asserting assertion; {for procedures}
return (expression) asserting assertion; {for functions}

All labeled statements must be assert statements, i.e., there must be an assertion at every label. The parentheses on the returned expression in functions are required. Return statements are mandatory since the Pascal construct

functionName := expression;

is not supported. All assertions are optional (although it may be difficult to verify programs without them).

8.2. Reading Programs: the readp command

To read and parse a program from the file A.B use

readp A.B;

An alternative is to say

readp;

to which the system asks

Input file:

to obtain the file name. The names of the *units* of the program are printed and each of the units is type-checked. The term *unit* means each of the procedures and functions, and the main program.

---

readp *fileName*;

file *fileName* should contain a Pascal program, consisting of a series of procedure or function definitions, and possibly a main program. The program units are read, parsed, and type-checked. Any Pascal programs, procedures, or functions read by previous readp commands *are forgotten*.

8.3. Generating Verification Conditions: the genvcs command

To generate verification conditions for, say units R1 and S2, use

genvcs R1, S2;

In general the command is genvcs, followed by a list of unit names separated by commas.[29]

Verification conditions may be generated or regenerated serially for one or more units. However, a second readp will cause the unit names obtained from the first readp to be lost, although the verification conditions for these units are retained. Thus the user is advised to read the entire program with a single readp command.

---

genvcs *PascalUnitNames*;

The *PascalUnitNames* must be names of programs, procedures, or functions read in by the *most recent* readp command. The control paths of the Pascal units associated with each name are determined, and for each path a verification condition (proposition) is generated. The set of verification conditions for each unit constitute the children of a shallow proof tree with root "verification(*unitName*)". Each verification condition is given a theorem name "*unitName*# *number*" [§7.3], where *number* is a small integer. The theorem names for a unit named *SimpleSend* would be "SimpleSend#1", "SimpleSend#2", etc. In addition, each verification condition is also given an arc label "VC*number*" [§7.6.2.3].

---

[29]The Auto Mechanism provides the *AutoGenvcs* profile entry for automatically performing the cases command after a readp command [§3.13].

8.4. Verification Condition Generation: Overview and Elementary Statements

The verification condition generator is a straightforward implementation of a set of axiomatic rules expressing the definition of the language in terms of a predicate transformer similar to Dijkstra's weakest liberal precondition transformer [Dijkstra 76]. Assignment, compound, conditional, and iteration statements are treated in standard ways. Thus, the assignment statement transformer is the substitution of the right-handside -expression for the left-hand-side variable. The substitution rules used in assignment statements preclude the presence of aliases since different formal or imported parameters are assumed to be different variables by the substitution rules.

The compound statement transformer is the composition of the transforms of the constituent statements. Conditional statements produce separate verification conditions for each possibility: if-then-else produces two, one assuming the Boolean condition (i.e., the then choice) and the other assuming the negation of the Boolean condition (i.e., the else choice); case produces a separate verification condition for each explicit choice plus a final one for the else choice. Iteration statements produce a verification condition for each path through the loop. For example, a while statement will in general produce three verification conditions--one from the entry to the loop invariant, one around the loop from the invariant back to the invariant, and one from the invariant to the loop exit.

8.4.1. Procedures

A procedure declaration produces verification conditions for the body of the procedure as well as producing the *computes-lemma*. To explain the computes-lemma, suppose a procedure heading is declared as follows:

```
procedure proc(var x: Integer; y: Integer)
   pre P(x, y);
   post Q(x, x', y);
```

where x' denotes the initial value of x. The computes-lemma produced is

```
theorem computesproc,
  all x, y, x1 (P(x, y) ∧ computes(proc(x, y), result(x1)) ⊃ Q(x1, x, y))
```

where x1 is a fresh variable. The computes-lemma records the procedure call in one hypothesis, the precondition as another hypothesis, and the postcondition as the conclusion. The variables of the precondition and postcondition are related via the computes-lemma. The computes-lemma need not be proved; it is the expression of the effects of calling the procedure and may be assumed, provided the body of the procedure is verified. When the procedure *proc* is called, say *proc*(z, w), the conjunct

computes(*proc*(z, w), result(z1))

where z1 is also a fresh variable, will be added as an hypothesis to the verification condition of the program path containing this call of *proc*. To make use of the hypothesis containing the computes-lemma at the appropriate point in the proof of the verification condition, the computes-lemma is instantiated in the same manner as is any lemma. In particular, the proper instantiation of variables

[Chapter 6] will provide the means of associating the variables of the call with the result variables. Finally, by discharging the precondition, one is able to use the postcondition in proving the verification condition.

8.4.2. Functions

For functions,[30] a similar idea of obtaining the result of a function call is used. Since no var variables are permitted in functions, no variables are set by a function call, and hence no renaming using fresh variables is necessary. There is one new idea, however. Since properties of a function can be stated and proved incrementally or by means of a collection of axioms about a whole set of functions, it is necessary to prove consistency of the function's properties. Given the declaration

```
function func(x: Integer) returns y: Integer;
    pre  P(x);
    post Q(x, y);
```

a verification condition of the form

$$\forall x (\exists y (P(x) \supset Q(x, y)))$$

is produced. This verification condition expresses the fact that it is possible to produce a result y from calling *func*(x).

8.5. A Formal Definition of Verification Condition Generation

Examples of generated verification conditions for most statement types are contained in Appendix IV. The verification conditions for a procedure of the form

```
procedure name(var j: type1; k: type2) imports (var m: type3; n: type4);
pre  P(j, k, m, n);
post Q(j, k, m, n, j1, m1);
declarations;
begin S end
```

are contained in the set of First Order Predicate Calculus wffs:

VC(assume P; S, Q) $\cup$ {computes-lemma}[31]

The functionality or interface of the verification condition generator is

VC(program with assertions, Boolean): set of verification conditions

where VC is defined below. Any assertion in S (and Q) can contain primed variables, which refer to the values held by the unprimed variables at the start of the procedure.

[30] As of ***Affirm*** version 1.21, the verification condition generator does not process functions correctly.

[31] The computes-lemma is immediately assumed, and is of the form

$(P(j, k, m, n) \wedge \text{computes}(\textit{name}(j, k, m, n), \text{result}(j1, m1))) \supset Q(j1, k, m1, n, j, m)$

This lemma should be used whenever a call to *name* occurs. See the rule for procedure call.

Notation

a, b, i are enumerated type expressions which do not change variables.

B refers to a Boolean expression in Pascal acceptable in the specification language. B does not modify any variables.

c_i refers to an element of a list of scalar constants.

C, Inv, P, Q
refer to logical assertions in the specification language.

E refers to a scalar expression in Pascal. E does not modify any variables.

Empty
refers to the null sequence of statements.

g is a label.

j, k, m, n, x
are lists of program variables.

j1, m1, x1
are fresh lists of variables.

$S, S_1, \ldots, S_n$
refer to (modified) Pascal statement lists.

x' referenced only in the section on **Complex Loops**, this symbol denotes the initial values of the associated program variables x upon entry to the loop.

y is a program variable.

Empty Statement and Semicolon

VC(Empty, Q) = {Q}

VC(S; ; , Q) = VC(S, Q)

Assignment Statement

VC(S; y := E(y, x), Q(y)) = VC(S, Q(E(y, x)))

Conditional Statements

$\mathrm{VC}(S;\ \underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2,\ Q)$
$= \mathrm{VC}(S;\ \underline{\text{assume}}\ B;\ S_1,\ Q)$
$\cup\ \mathrm{VC}(S;\ \underline{\text{assume}}\ {\sim}B;\ S_2,\ Q)$

$\mathrm{VC}(S;\ \underline{\text{case}}\ E\ \underline{\text{of}}\ c_1: S_1; \ldots ; c_n: S_n\ \underline{\text{else}}\ S_{n+1}\ \underline{\text{end}},\ Q)$
$= \cup_{1 \leq i \leq n} \mathrm{VC}(S;\ \underline{\text{assume}}\ E \in c_i;\ S_i,\ Q)$
$\cup\ \mathrm{VC}(S;\ \underline{\text{assume}}\ E \notin \cup_{1 \leq i \leq n} c_i;\ S_{n+1},\ Q)$

Assertion Statements

$VC(S; \underline{assert}\ B, Q) = VC(S, B) \cup \{B \supset Q\}$

$VC(S; \underline{prove}\ B, Q) = VC(S, B) \cup VC(S, B \supset Q)$

$VC(S; \underline{assume}\ B, Q) = VC(S, B \supset Q)$

Goto Statement

$VC(S; \underline{goto}\ g, Q) = VC(S, B)$ where B is the assertion at label g.

All labeled statements must be $\underline{assert}$ statements.

Procedure Call Statement

$VC(S; \mathit{name}(j, k), Q)$

$Q(j1, m1)$

$= VC(S; \underline{assume}\ computes(\mathit{name}(j, k, m, n), result(j1, m1)), Q)$

Simple Loop Statements

$VC(S; \underline{maintain}\ Inv(x)\ \underline{while}\ B(x)\ \underline{do}\ S_1(x), Q(x))$

$= VC(S, Inv(x))$

$\cup\ VC(\underline{assume}\ Inv(x) \wedge B(x); S_1(x), Inv(x))$

$\cup\ VC(S, Inv(x1) \wedge \sim B(x1) \supset Q(x1))$

$VC(S; \underline{maintain}\ Inv(x, i)\ \underline{for}\ i := a\ \underline{to}\ b\ \underline{do}\ S_1(x, i), Q(x))$

```
= VC(S; i := a; maintain Inv(x, i) while i ≤ b do begin
                                       assume i ≥ a;
                                       S_1(x, i);
                                       i := i + 1 end,
        Q(x))
```

$= VC(S, Inv(x, a))$

$\cup\ VC(\underline{assume}\ Inv(x, i) \wedge a \leq i \leq b; S_1(x, i), Inv(x, i+1))$

$\cup\ VC(S, Inv(x1, i) \wedge i > b \supset Q(x1))$

Appropriate changes are necessary for other enumerated types.

$VC(S; \underline{repeat}\ S_1(x)\ \underline{maintain}\ Inv(x)\ \underline{until}\ B(x), Q(x))$

$= VC(S; S_1(x); \underline{maintain}\ Inv(x)\ \underline{while} \sim B(x)\ \underline{do}\ S_1(x), Q(x))$

$= VC(S; S_1(x), Inv(x))$

$\cup\ VC(\underline{assume}\ Inv(x) \wedge \sim B(x); S_1(x), Inv(x))$

$\cup\ VC(S; S_1(x), Inv(x1) \wedge B(x1) \supset Q(x1))$.

Complex Loop Statements

```
VC(S; maintain Inv(x', x) while B(x) do S_1(x) thus C(x', x), Q(x))
        = VC(S, Inv(x, x))
        ∪ VC(x' := x; assume Inv(x', x) ∧ B(x); S_1(x),
              Inv(x', x)
              ∧ [~B(x1) ∧ Inv(x, x1) ∧ C(x, x1)
                      ⊃ Inv(x', x1) ∧ C(x', x1)])
        ∪ {Inv(x, x) ∧ ~B(x) ⊃ C(x, x)}
        ∪ VC(S, Inv(x, x1) ∧ ~B(x1) ∧ C(x, x1) ⊃ Q(x1))

VC(S; maintain Inv(x', x, i) for i := a to b do S_1(x, i) thus C(x', x, i),
    Q(x))
        = VC(S; i := a; maintain Inv(x', x, i)
                while i ≤ b do begin assume i ≥ a;
                                  S_1(x, i); i := i + 1 end
                thus C(x', x, i), Q(x))
```

Appropriate changes are necessary for other enumerated types.

```
VC(S; repeat S_1(x) maintain Inv(x', x) until B(x) thus C(x', x), Q(x))
        = VC(S; S_1(x); maintain Inv(x', x)
                        while ~B(x) do S(x)
                        thus C(x', x), Q(x))
```

9. The Formula IO Machine

9.1. Introduction

The Formula IO Machine contains the facilities necessary to read, list, and/or save ***Affirm*** objects such as type specifications, command files, and ***Pascal*** programs. As of ***Affirm*** version 1.21, this machine is quite incomplete, in that the user has little or no control over most portions of input/output. In addition, there is no way of listing specific components of a complex object without listing the entire object. There is nothing like writing a reference manual to determine what is missing from a large system!

The following sections describe the commands which read, list, and save several of the objects upon which ***Affirm*** works, in particular type specifications. At this point, the provided data base facilities are quite limited. The load, save, compile, and freeze commands (all described below) are quite useful for saving and re-loading type specifications or saving the state of the entire system. The needs command is quite useful for documenting the dependencies of one type on the other types it uses. This command ensures that the types provided to it as its parameter are each loaded or read before processing the remainder of the specification of the current type.

9.2. The Transcript File and the transcript command

Affirm automatically opens a *transcript* file when it begins. This transcript file contains a nearly verbatim echo of all input and output, and acts as a history of the session. The transcript file name is governed by the profile entry *TranscriptFileName*. The transcript command can be used to open a different transcript, or to turn the transcript mechanism off.

---

transcript [*fileName*];

begins a (new) transcript file *fileName*. If there is no transcript file at the time the user issues this command, then the file name of the new transcript, if not provided in the command, is governed by the profile entry *TranscriptFileName*. If there is a transcript file at the time this command is issued, then the new file name, if not provided in the command, is identical to the old filename, with a new version number. The transcript file when the system first begins is written into the user's login directory, rather than the connected directory. Later transcript commands default to the connected directory.

transcript off;

turns off the transcript. *Not recommended.*

9.3. Proposition Listing

The user has little control over the format of the proposition listed during theorem proving after each normalization [§6.3.2].[32] The user is cautioned against cutting up a transcript in some text editor and then attempting to re-read a proposition. The main problem is the mechanism for showing the precedence of binary infix operators. The printer often shows precedence implicitly, via alignment and spacing. The parser ignores formatting, and relies on explicit parenthese and assumed precedence (as shown on page 65).[33]

9.4. Type Specification Listing

For the most part, ***Affirm*** can re-read the output generated by the print command with the type option, as for example in the command

```
print type SequenceOfElemType;
```

The user can edit the transcript file, and then read the type specification back in, with one major exception:

- Operator precedence of binary infix operators is somewhat of a problem. User-defined binary infix operators all have a specific priority, so the user is advised to add parentheses liberally.

9.5. The needs Command: A Primitive Type Database Facility

Data types are usually defined in terms of other data types. If these other types are not loaded, ***Affirm*** will stop and ask the user to define them. The system also saves a list of the data types needed by the one currently being defined, and automatically searches for the specifications of these data types the next time the current type is loaded or read. This mechanism is explicitly available to the user via the needs command. A set of file name conventions is used to find the files containing the data type specifications, as follows. A file name on ***Tops-20*** and ***Tenex*** consists of three fields: a *name*, an *extension*, and a *version* number. In our file naming conventions, the name field is simply the type name (in upper case). The version field is not explicitly used, so that it retains its meaning to the underlying file system: the greatest number is considered the most recently written file, etc. The extension field is used to further describe the contents of the file, as follows.

AXIOMS
The source text of the data type specification, to be read by ***Affirm***.

empty
The saved version of the specification, to be loaded by ***Affirm***.

COM The compiled version of the specification, to be loaded by ***Affirm***.

---

[32]There are several pertinent profile entries [§3.13] affording some control. These include *TerminalLineWidth*, *NewPP*, *AverageNameLength*, and *UseOrInProps*.

[33]The Users Guide provides some pointers on how to avoid most of this sort of problem.

FANCY-AXIOMS

A version of the source text containing fonting commands for pretty output using ***SCRIBE***.[34] This version can not be read or loaded by ***Affirm***!

For example, the file named SETOFELEMTYPE.AXIOMS is assumed to contain the source text of the data type specification for the type *SetOfElemType* (or any type spelled with those characters, in either casing).

The underlying Interlisp system keeps track of objects that have a file associated with them (such as data types in ***Affirm***). Since each type has an associated file, and file names are uppercased versions of the type name,

- Type names may not differ only in casing.

9.6. General File IO

read [*fileName*];

causes ***Affirm*** to read *fileName*. The file must contain ***Affirm*** commands. The last command in the file must be the stop command. *FileName* is a normal text file that the user presumably created using some text editor.

readp [*fileName*];

causes ***Affirm*** to read *fileName*. The file must contain ***Pascal*** programs [§8.2]. *FileName* is assumed to be a normal text file.

load [*fileName*];

causes ***Affirm*** to load file *fileName*. The file must have been previously written using the save command. The only ***Affirm*** object which can be saved and then loaded is a data type specification. Note that the file contents are *not* normal text, and cannot be directly modified by the user.

save type *typeName*;

causes ***Affirm*** to write a file containing the specification of the type. The file name of the file is the upper-case version of the type name. The save command can be used in conjunction with the load command to remember data type specifications across ***Affirm*** sessions. The file written by the save command for types contains the internal form of the type specification (Interlisp code). Thus little processing is required to load the type back into ***Affirm***, compared to the processing required when first creating the specification.

compile type *typeNames*;

writes a file containing a compiled version of the internal representation of a data type specification (Interlisp code). All stable types should be compiled, since this form of the type uses the least space and runs the fastest. Any types still undergoing development should be saved, rather than compiled.

[34] A document production system developed by the Computer Science Department of Carnegie-Mellon University, and now distributed by UNILOGIC, Pittsburgh.

print file [*fileName*];

causes ***Affirm*** to copy the contents of file *fileName* to the terminal (and transcript, if there is one). This is useful for documentation purposes.

stop; should be used only in a file of ***Affirm*** commands, as the last command. It avoids the usual end-of-file problems.

freeze [*fileName*];

causes the entire system state[35] to be written into file *fileName*. The default freeze file name when none is provided in the command is determined by the user profile entry *FreezeFileName*. The size of the file written is on the order of 300 pages. This file can then be run at a later time by simply typing the file name at the operating system executive level. The user will then be back in ***Affirm*** at the executive, as if the freeze had never happened (except that a new transcript file will be opened, if necessary). This command is quite useful for freezing a session in place, and then continuing it later. (Compare this with the save command, which does not save the entire system, but just relatively small components of it.)

needs type[s] $typeName_1$ [, ..., $typeName_n$];

should be used immediately after a type command, before any other part of the type specification. This command ensures that all the types $typeName_i$ for $1 \leq i \leq n$ are either loaded or read, before any more of the specification of the current type is processed. If type $typeName_i$ is already defined, it is not re-defined. If $typeName_i$ is not yet defined, then the most recent version of its specification is found. The algorithm that finds the files containing the types to be input searches a set of directories for the most recent version of the specification of each type, whether that version be in original source form or in the internal saved form, or even in compiled form. For each type requiring such a directory search, ***Affirm*** first identifies the possible set of files containing versions of the type specification; it then ranks the versions (by using the file write date to determine which file was most recently written). ***Affirm*** will then proceed to load or read that file, as is appropriate. (The set of directories used as of ***Affirm*** version 1.21 is {*connected*, *login*, PVLibrary, Affirm}.)

[35]The freeze command does not save the state of any open files.

Appendix I
The Syntax of User Commands

The grammatical presentation method used here was designed by David Wile [Wile 79]. In this scheme, terminal symbols are prefixed with a single quote, and are displayed in a `typewriter-like font`. Nonterminal symbols are simple identifiers, and are displayed in *italics*. The form

symbol1 ↑ *symbol2*

means

One or more occurrences of *symbol1*, separated by *symbol2*.

For example,

id ↑ ',

represents a list of identifiers separated by commas. The form

[*symbolSequence*]

means

Zero or one occurrences of *symbolSequence*.

The empty string is denoted by ε, the Greek letter epsilon.

Commands most likely to be needed by an inexperienced user are marked in the left margin with the symbol "**".

Other conventions should be obvious.

```
        AffirmCommand := ';
                |'@ [ arbitraryTextExceptSemicolon ] ';

**              |'abort';
                |'adopt typeName ';
**              |'annotate [ nodeName ', ] arbitraryTextExceptSemicolon ';
**              |'apply [ nodeName ', ] proposition ';
                |'arc arcLabel ';
                |'assume [ nodeName ', ] proposition ';
                |'augment proposition ';
**              |'axiom rule ';
                |'axioms rule ↑ ', ';

                |'cases ';
                |'choose number ↑ ', ';
                |'clear 'proof ';
                |'compile objects ';
                |'complete ';

**              |'declare id ↑ ', ': typeName ';
**              |'define rule ↑ ', ';
                |'denote ( expression 'by variable ) ↑ ', ';
                |'discard objects ';
                |'down child ';

                |'e InterlispCommand
**              |'edit typeName ';
**              |'employ schemaName '( allVariable ') ';
**              |'end ';
                |'eval expression ';
                |'exec ';

**              |'fix [ eventSpecification ] ';
**              |'freeze [ fileName ] ';

                |'genvcs procedureName ↑ ', ';
**              |'gripe shortTitle ';

                |'infix interfaceName ↑ ', ';
**              |'interface lhs ': typeName ';
                |'interfaces lhs ↑ ', ': typeName ';
**              |'invoke rangedExp ↑ ', ';

                |'let instantiation ↑ ', ';
                |'lisp ';
                |'load [ fileName ] ';
        |'name nodeName [ ', proposition ] ';
**      |'needs objects ';
**      |'next ';
        |'normalize ';
        |'normint ';
**      |'note arbitraryTextExceptSemicolon ';

**      |'ok ';

**      |'print printOptions ';
**      |'profile [ transaction ↑ ', ] ';
**      |'put instantiation ↑ ', ';

**      |'quit ';

**      |'read [ fileName ] ';
        |'readp [ fileName ] ';
        |'redo [ eventSpecification ] ';
**      |'replace [ expression ↑ ', ] ';
        |'resume ';
        |'retry ';
        |'review ';
**      |'rulelemma rule ';
        |'rulelemmas rule ↑ ', ';

**      |'save objects ';
**      |'schema rule ';
        |'schemas rule ↑ ', ';
**      |'search ';
        |'set variableName 'to expression ';
        |'stop ';
        |'storage ('normal |'severe |'tight ) ';
        |'sufficient? [ typeName ] ';
**      |'suppose [ expression ] ';
        |'swap rangedExp ↑ ', ';

        |'thaw [ fileName ] ';
        |'theorem [ nodeName ', ] proposition ';
        |'transcript [ 'on |'off | fileName ] ';
**      |'try [ nodeName ', ] proposition ';
**      |'type id ';

**      |'undo [ eventSpecification ] ';
        |'up [ number ] ';
        |'use [ nodeName ', ] proposition ';

        | undocumentedAffirmCommands
        ;
```

```
allVariable := id ;

child :=
          arcLabel
        | nodeName
        | ordinalInteger ;

coord :=
          number
        | '- number
        | 'ALL
        | 'LAST
        | 'FIRST ;

definedName := id ;

elementName := id ;

eventSpecification :=
          number
        | AffirmCommandName ;

expression := primary [ infixOp expression ] ;

infixOp :=
          '~ '=          | '! '=
        | '< '=
        | '> '=
        | userDefinedOp ;

instantiation := someVariable '= expression ;

interfaceName := id ;

lhs :=
          interfaceName [ '( expression ↑ ', ') ]
        | expression interfaceName expression ;

nodeName :=
          id
        | number ;
```

```
objectName :=
        'AffirmObjects
       |'Arcs
       |'Axioms
       |'Commands
       |'Definitions
       |'Directories
       |'Disconnected
       |'Files
       |'FileTypes
       |'Groups
       |'HelpTopics
       |'History
       |'Interfaces
       |'Lemmas
       |'Lhs
       |'Nodes
       |'PrintObjects
       |'ProfileEntries
       |'Schemas
       |'Theorems
       |'TypeParts
       |'Types
       |'Variables
       ;

objects := objectName ( elementName | lhs ) ↑ ', ;

opOrExpression :=
        expression
       | infixOp
       | prefixOp ;

prefixOp := '~          |'not
       | userDefinedOp ;

primary :=
        prefixOp [ '( expression ↑ ', ') ]
       | variable
       | number
       | prefixOp primary
       | '( expression ')
       | 'if expression 'then expression [ 'else expression ]
       | quantifier identifier ↑ ', '( expression ') ;
```

```
printOptions :=
        '?
       |'assumptions
       |'BadEquations
       |'both [printOptions2]
       |'file fileName
       |'history
       |'IH
       |'known objectName
       |'named
       |'names
       |'next
       |'original
       |'proof [printOptions2]
       |'prop [printOptions2]
       |'result
       |'status [printOptions2]
       |'type typeName [typeParts]
       |'unproven
       |'uses [printOptions2]
       |'variable
       |'variables;

printOptions2 := ['list|'nolist]('T|'*|'theorem|'unproved) nodeName;

procedureName := id;

profileEntryName := id;

proposition :=
        expression
       |nodeName;

quantifier :=
        'all
       |'some;

range := coord [': coord];

rangedExp := opOrExpression [rangeSpec];

rangeSpec := '| range↑', '|;

rule := lhs '=' expression;

schemaName := id;

someVariable := id;
```

transaction := *profileEntryName* [('? | '= *profileValue*)] ;

typeName := *id* ;

typeParts :=

```
        'axiom       |'axioms
       |'declare
       |'define      |'defn.
       |'interface   |'interfaces
       |'needs
       |'rulelemma   |'rulelemmas
       |'schema      |'schemas
       ;
```

*undocumented***Affirm***Commands* :=

```
        'batch ['on |'off ]';
       |'monitor && ';
       ]'renumber [ eventSpecification ]';
       | InterlispCommand
       ;
```

userDefinedOp := *interfaceName* ;

variable := *id* ;

Appendix II
The Syntax of Extended Pascal

This appendix contains a grammar of the programming language processed by ***Affirm***. The grammatical presentation method was previously described in Appendix I.

```
program := ( procedureOrFunctionDeclaration | block ) [ '; ] [ '. ] ;

arrayType := 'array '[ simpleType ↑ ', '] 'of type ;

assertion := expression ;

assertStatement := 'assert assertion ;

assignmentStatement := variable ': '= expression ;

assumeStatement := 'assume assertion ;

block := [ ( 'entry | 'pre ) assertion '; ]
    [ ( 'exit | 'post ) assertion '; ]
    [ declareopt ↑ '; '; ]
    [ compoundStatement ] ;

bracketExprList := '[ [ expression ↑ ', ] '] ;

caseElementList := caseLabel ↑ ', ': [ statement ] ;

caseLabel := constant ;

caseStatement := 'case expression 'of caseElementList ↑ ';
    [ '; ( 'else | 'otherwise ) statement ] [ '; ] 'end ;

compoundStatement := 'begin statement ↑ '; 'end ;

concurrentAssignmentStatement := variable ↑ ', ': '= expression ↑ ', ;

constDefinition := identifier '= expression ;

declareopt := [ 'xpublic | 'public ] declareType ;

declareType := 'label label ↑ ', ';
    | 'const constDefinition ↑ ';
    | 'type typeDefinition ↑ ';
    | 'var varDeclaration ↑ ';
    | procedureOrFunctionDeclaration ;

direction := 'to
    | 'downto ;
```

```
expression := primary [ infixOp expression ] ;

expressionSeq := expression ↑ ', ;

fieldList := [ recordSection ↑ '; ] [ '; variantPart ] [ '; ] ;

fileType := 'file 'of type ;

formalParameterSection := [ parameterKind ] parameterGroup ;

forStatement := [ 'maintain assertion ]
    'for identifier ': '= expression direction expression
    'do statement [ 'thus assertion ] ;

functionDecl := expression ↑ ', ': expression ;

goToStatement := ( 'goto | 'go 'to ) label [ 'asserting assertion ] ;

greaterThanEqual := '> '= ;

ifExpr := 'if expression 'then expression [ 'else expression ] ;

ifStatement := 'if expression 'then statement [ 'else statement ] ;

infixOp := notEqual
    | lessThanEqual
    | greaterThanEqual
    | normalInfixOp ;

label := && ;

labelStatement := label ': [ simpleStatement ] ;

lessThanEqual := '< '= ;

normalInfixOp := && ;

notEqual := '~ '= | '! '= ;

packed := 'packed ;

parameterGroup := identifier ↑ ', [ ': type ] ;

parameterKind := 'var
    | 'function
    | 'procedure ;

parenExpr := '( expression ') ;
```

```
pointerType := '↑ identifier ;

prefixExpr := prefixOp '( [ expression ↑ ', ] ')
    [ 'imports '( identifier ↑ '; ') ] ;

prefixOp := && ;

primary := prefixExpr
    | variable
    | number
    | specialPrefixExpr
    | parenExpr
    | bracketExprList
    | ifExpr
    | quantifiedExpression ;

procedureOrFunctionDeclaration := [ 'inline ] unitKind identifier
    [ '( formalParameterSection ↑ '; ') ]
    [ 'returns ] [ identifier ] [ ': type ]
    [ 'imports '( formalParameterSection ↑ '; ') ]
    [ 'alters identifier ↑ ', ] '; block ;

procedureStatement := identifier [ '( expression ↑ ', ') ]
    [ 'imports '( identifier ↑ '; ') ] [ 'alters variable ↑ ', ] ;

proveStatement := 'prove assertion ;

quantifiedExpression := quantifier identifier ↑ ', '( expression ') ;

quantifier := 'all
    | 'forall
    | 'some
    | 'exists ;

recordSection := identifier ↑ ', ': type ;

recordType := 'record fieldList 'end ;

repeatStatement := 'repeat statement ↑ '; 'until expression
    [ 'thus assertion ] ;

returnStatement := 'return [ '( expression ') ] [ 'asserting assertion ] ;

scalarType := '( identifier ↑ ', ') ;

setType := 'set 'of simpleType ;
```

```
simpleStatement := compoundStatement
    | ifStatement
    | caseStatement
    | whileStatement
    | repeatStatement
    | forStatement
    | withStatement
    | goToStatement
    | assertStatement
    | returnStatement
    | proveStatement
    | assumeStatement
    | assignmentStatement
    | concurrentAssignmentStatement
    | procedureStatement ;

simpleType := scalarType
    | subrangeType
    | typeIdentifier ;

specialPrefixExpr := specialPrefixOp primary ;

specialPrefixOp := && ;

statement := assignmentStatement
    | labelStatement
    | simpleStatement
    | ε ;

structuredType := [ packed ] unpackedStructuredType ;

subrangeType := ( '* | expression ) '. '. ( '* | expression ) ;

type := simpleType
    | structuredType
    | pointerType ;

typeDefinition := identifier '= type ;

typeIdentifier := identifier ;

unitKind := 'procedure
    | 'function
    | 'program ;

unpackedStructuredType := arrayType
    | recordType
    | setType
    | fileType ;
```

varDeclaration := *varDeclarePart* ↑ ', ': *type* ;

varDeclarePart := *identifier* ['@ *expression*] [': '= *expression*] ;

variable := *identifier* ;

variableDecl := *identifier* ↑ ', ': *expression* ;

variant := [*caseLabel* ↑ ', ': '(*fieldList* ')] ;

variantPart := 'case [*identifier* ':] *typeIdentifier* 'of *variant* ↑ '; ;

whileStatement := ['maintain *assertion*] 'while *expression* 'do *statement*
['thus *assertion*] ;

withStatement := 'with *variable* ↑ ', 'do *statement* ;

Appendix III
Affirm's Interactions with Interlisp

Affirm is implemented in Interlisp [Teitelman 78] and attempts to use the power of the provided environment rather than completely re-creating features already present in Interlisp. For example, the event history window is really Interlisp's history list. This appendix highlights the dependencies on Interlisp of which the user (casual or otherwise) must currently be aware in using ***Affirm***.

III.1. Obtaining Access to Interlisp

Affirm currently has two commands providing access to the Interlisp system. Provision of such access is of course accompanied with the warning that soundness of any proof is highly questionable if, in the process of proving, the user has jumped into Interlisp and fiddled. Such access is provided because the system is *experimental*.

The two commands differ in that the first, e, performs *one* Interlisp command, and then returns to ***Affirm***'s executive, while the second, lisp, drops the user into the Interlisp system (at its command interpreter), from which the user must explicitly return to the ***Affirm*** executive by typing OK to the Interlisp command interpreter.

If you modify *any* of ***Affirm***'s data structures or functions, you'll get what you deserve, and undoubtedly sooner than you expect.

III.2. The Interlisp Editor

Affirm uses the Interlisp editor in the implementation of the @ command [§III.3]. We don't recommend it use, it was necessary in older version of ***Affirm***. This section contains a short summary of the Interlisp editor. The entire discussion can be found in [Teitelman 78;ch-9].

The Interlisp editor provides a convenient means of modifying list structures. It is a structure editor: the user is moving around the expression as if it were a tree. The editor maintains a *current expression*, or *focus of attention* within the expression that is being edited. We shall use these two terms interchangeably. Initially, the focus of attention is the entire expression. Some of the commands that are useful in moving about within the expression structure are:

n — *n* is a positive integer. This command moves the focus of attention to the nth element of the current expression. Caution: the command (*n*) deletes the nth element.

F *pattern*
: The F command attempts to find *pattern* within the current expression. If this search is successful then the focus of attention becomes the expression that matches *pattern*. *Pattern* can be any atom, and can contain escapes (which the operating system indicates as $). Each escape can match zero or more contiguous characters in an atom, e.g., VER$ matches VERYLONGATOM. The command will print a message if it cannot find the pattern.

NX — This command moves the focus of attention to the next sibling. For example, if the expression being edited is

```
(PLUS (FOO 2) (FUM 3))
```

and the current expression is

```
(FOO 2)
```

then the NX command would focus upon

```
(FUM 3)
```

This command is very useful after the user uses the *n* command and then discovers that he or she mis-counted.

BK This command modifies the current expression to be the previous sibling if possible.

!0 This command modifies the focus of attention to be the parent of the current expression.

↑ This command resets the focus of attention to the entire initial expression.

P This command prints the current expression, showing the structure, (but not the contents) of contained subexpressions, a few levels deep.

PP This command pretty-prints the current expression.

III.3. Subexpression Specification: The @ Command

The @ command invokes the Interlisp editor with Current Proposition as its argument. Therefore, this command can be used in conjunction with the Interlisp editor commands [§III.2] to delimit subexpressions of the current proposition.

The @ command has a series of subcommands, consisting of a subset of top-level ***Affirm*** commands and Interlisp editor commands.

The list structure containing Current Proposition is in *prefix* form. For example, the expression

```
IH(q) imp subseq(q, q apr x)
```

is stored internally as

```
(imp (IH q) (subseq q (apr q x)))
```

There are two important points we wish to emphasize:

1. There is no supervision by ***Affirm*** of what the user does in the editor. In order to preserve soundness, the user is enjoined not to delete elements of the theorem. Only attention-changing commands (such as F, 0, 2, NX, !0, and ↑) and those listed below should be used.

2. When using the editor, the names typed by the user are not automatically lengthened to their internal form. Therefore, when performing an F (find) editor command, the user is advised to end the pattern with an escape character.

The Interlisp editor commands useful to the user are documented in Section III.2. The ***Affirm*** subcommands are described here. The subcommands of the @ command are as follows:[36]

5 2 This is really just two instances of the *n* command [§III.2], but is described here because the two integers, 5 and 2, are dependent on the internal data structure containing the proposition. This sequence of Interlisp editor commands modifies the focus of attention to be the sequence of hypotheses of Current Proposition.

5 3 As described above, this sequence of Interlisp commands modifies the focus of attention to the conclusions of Current Proposition.

(invoke *definedName*)
The first instance of the definition with name *definedName* in the current expression is expanded.

(delete *n*)
The n_{th} element of the current expression is deleted.

(delete n_1 n_2 n_3)
The children at the listed positions are deleted. These indices are *instantaneous*, not *one-at-a-time*.

(extract *n*)
The current expression is replaced with its n_{th} child. For example, if the current expression is `(AND e1 e2 e3)` then

The command:	will result in:
(delete 2)	`(AND e2 e3)`
(delete 2 3)	`(AND e3)`
(extract 2)	`e1`

It is not sound to delete operators.

Pa, PPa, (Pa --)
These are just like their Interlisp counterparts P and PP [§III.2], except that they print names in a nicer form.

infix The current expression is printed in infix form.

eval The current expression is evaluated.

ok The user is returned to the ***Affirm*** executive, and the modified expression becomes Current Proposition.

stop The edit is aborted; no changes are made to Current Proposition, and the user is returned to the ***Affirm*** executive.

[36] Note *and beware!* that the syntax given here differs markedly from the normal command syntax.

III.4. Errors and Breaks

While it is not a *common* experience, it *can* happen that the user will stumble across one of the 'undocumented curious features' that cause ***Affirm*** to drop the user into the *break package* of Interlisp, usually when the user is least expecting it. The user can prevent any unexpected access to Interlisp by using the profile entry *BreakAccess* [§3.13]. The first questions to ask are "When am I talking to Interlisp?" and "When I am talking to ***Affirm***?" If the system prompts the user with something like:

```
10_

10:

10*
```

where 10 could be any integer, then the user is communing with Interlisp and not with ***Affirm***. The colon prompt will arise from an Interlisp break, and the underbar prompt is displayed by the Interlisp interpreter. The asterisk is displayed by the Interlisp editor.

Inside an Interlisp break, the safest thing to do is to type an up-arrow (↑). This will, hopefully, return the user to the ***Affirm*** executive. The next best thing to do is type **control-D** (the panic interrupt for ***Affirm***). This will either return the user to the ***Affirm*** executive or leave the user at the Interlisp command interpreter with the underbar prompt (–). In the latter case, the user should then type

```
(AffirmExec)
```

If the user somehow ends up in the Interlisp editor unexpectedly, he or she should then type

```
STOP
```

Whether or not ***Affirm*** is sound after the user returns to the executive is a function of whatever caused the error. The best recourse would be to restart with a fresh ***Affirm***. One thing to always consider after a break is whether or not the transcript file [§9.2] has somehow been closed. The ***Affirm*** executive will print a message warning the user if the transcript has somehow been closed.

III.5. control-T: Finding Out What's Going On

At any time, the user can type **control-T** (in either ***Tops-20*** or ***Tenex***) to determine if the system is still responding when it seems a little slow. The **control-T** character is an Interlisp *immediate interrupt* character, so it does not interfere with any other input: it is acted upon immediately, and is never really seen by any of the normal input routines.

III.6. Control Character Definitions

The following characters have special meaning to Interlisp and should not be typed unless the user wants the resulting action.

control-A
: On ***Tops-20*** it is used by the Interlisp editor and should not be typed by the user. On ***Tenex***, this backs up one character. If there are no more characters on the line, the bell is rung.

control-B
: ***Affirm*** users should not type this character. Computation is stopped, the stack is backed up to the last function call, and a break occurs.

control-C
: Computation is stopped by the operating system when this character is typed. In order to continue with no ill effects, the user should type the operating system command CONTINUE. The point at which the control-C takes effect depends on the current state of ***Affirm*** and how many control-C's are typed. Two control-C's will stop ***Affirm*** immediately. One control-C will stop ***Affirm*** as soon as the control-C is read.

control-D
: control-D immediately stops the computation and returns control to the command interpreter of Interlisp or to the ***Affirm*** top-level executive, depending on the value of the profile entry *BreakAccess*.

control-E
: This aborts any ***Affirm*** command. An attempt is made to restore the state of the computation, just as if the command currently being executed had not occurred. This undoing works most of the time, but under certain circumstances will not undo the last command and ***Affirm*** will be left in an unsound state.

control-F
: This is used by the operating system to recognize one field when typing in file names.

control-G
: This rings the bell in the terminal.

control-H
: ***Affirm*** users should not use this character. It causes an Interlisp break at the next function call.

control-I
: control-I is a tab character. ***Affirm*** treats it like a blank.

control-J
: control-J is a linefeed. It is used by the Interlisp editor as a command.

control-K
: This aborts the printing of expressions.

control-L
: control-L is a formfeed. This is used by the Interlisp editor on ***Tops-20***.

control-M
: control-M is a carriage return.

control-N

Affirm users should not type this character. On ***Tops-20***, if typed in the middle of an expression being typed in, control-N will cause the Interlisp editor to be called on the expression when the expression is finished being typed in.

control-O
: This clears the output buffer. (Typically the output buffer is only about 20 characters.)

control-P
: ***Affirm*** users should not type this character. It changes printlevel.

control-Q
: On ***Tops-20*** it is used to resume frozen typeout, and on ***Tenex*** this is used to cancel the current line.

control-R
: This causes ***Affirm*** to retype the current input line.

control-S
: On ***Tops-20*** it is used to freeze typeout (resume typeout with control-Q). This causes Interlisp to change MINFS on ***Tenex*** and should therefore not be typed by ***Affirm*** users.

control-T
: This prints the status of ***Affirm***. It is useful for determining whether or not the system is waiting for input.

control-U
: On ***Tops-20*** control-U is used to cancel the current line. On ***Tenex Affirm*** users should not type this character. If it is typed in the middle of an expression being input, it will cause the editor to be called on the expression when it is finished being typed.

control-V
: The operating systems use this character as the quoting character. ***Affirm*** does not recognize this character as the quoting character; instead, it uses the percent sign.

control-W
: ***Affirm*** users should not type this character.

control-X
: This is active only during application of the Knuth-Bendix algorithm testing convergence of the set of rewrite rules in the system; upon receipt of control-X, ***Affirm*** will drop into a lower executive.

control-Y
: ***Affirm*** users should not type this character. It is a read macro for Interlisp.

control-Z
: On ***Tops-20*** this character clears the terminal input buffer. This is used by the Interlisp editor on ***Tenex***.

ESC control-[is another way to type this character. It is used by the operating systems to recognize a command or file. It is normally echoed as $.

DEL It is used by ***Tops-20*** to back up one character. This is not really a control character. It clears the terminal input buffer on ***Tenex***. Noise on a telephone line frequently sends out DELS.

III.7. Summary of Useful Control Characters

***Tops-20*:**

Most Useful:
C, E, R, S, T, U, DEL

Sometimes Useful:
D, F, K, X, Z, Escape

Rarely Useful:
A, B, G, H, I, J, L, M, N, O, P, V, W, Y

***Tenex*:**

Most Useful:
A, C, E, R, T

Sometimes Useful:
D, F, K, Q, X, DEL, Escape

Rarely Useful:
B, G, H, I, J, L, M, N, O, P, S, U, V, W, Y, Z

Appendix IV
Examples of Generated Verification Conditions

This Appendix contains a set of examples of verification conditions produced from program fragments. Each section below contains a program fragment and the verification conditions generated from it. Except for the use of fonts for display purposes, what is shown in each section is verbatim input and output.

IV.1. The Assert Statement

```
procedure testassert (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin
  assert B (i, j, i')
end;
```

```
    There are 2 verification conditions for testassert:

    testassert # 1 --     B(i, j, i')
              imp postc(i, j, i')


    testassert # 2 -- prec(i, j) imp B(i, j, i)
```

IV.2. The Assume Statement

```
procedure testassume (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin
  assume B (i, j, i')
end;
```

```
    There is 1 verification condition for testassume:

    testassume # 1 --     prec(i, j) and B(i, j, i)
              imp postc(i, j, i)
```

IV.3. The Assignment Statement

```
procedure testassign (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin
  i := 1
end;
```

```
    There is 1 verification condition for testassign:

    testassign # 1 -- prec(i, j) imp postc(1, j, i)
```

IV.4. The Case Statement

```
procedure testcase (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin
  case j of
    1:     i := 3;
    2, 3:  i := 4;
    4:     i := 17;
    else   i := 12
  end
end;
```

There are 4 verification conditions for testcase:

```
testcase # 1 --    prec(i, j) and (j = 1)
           imp postc(3, j, i)


testcase # 2 --    prec(i, j) and ((j = 2) or (j = 3))
           imp postc(4, j, i)


testcase # 3 --    prec(i, j) and (j = 4)
           imp postc(17, j, i)


testcase # 4 --        prec(i, j)
               and not  or (j = 1,
                           j = 2,
                           j = 3, j = 4)
           imp postc(12, j, i)
```

IV.5. The For Statement (Inductive Assertions)

```
procedure testfor₁ (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
var k: Integer;
begin
  maintain II (i, j, i', k)
  for k := 1 to 10 do
    i := i + j
end;
```

There are 3 verification conditions for testfor1:

```
testfor1 # 1 --        prec(i, j)
            and II(i2, j, i, k1)
            and not (k1 le 10)
         imp postc(i2, j, i)


testfor1 # 2 -- prec(i, j) imp II(i, j, i, 1)


testfor1 # 3 --        II(i, j, i', k)
            and k le 10
            and 1 le k
         imp II(i + j, j, i', k + 1)
```

IV.6. The For Statement (Subgoal Assertions)

```
procedure testfor2 (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
var k: Integer;
begin
  maintain IIF (i, j, i', k, iInitial)
  for k := 1 to 10 do
    i := i + j
  thus FW (i, j, i', k, iInitial)
end;
```

There are 4 verification conditions for testfor2:

```
testfor2 # 1 --       prec(i, j)
                and IIF(i2, j, i, k1, i)
                and not (k1 le 10)
                and FW(i2, j, i, k1, i)
            imp postc(i2, j, i)

testfor2 # 2 --    prec(i, j)
            imp IIF(i, j, i, 1, i)

testfor2 # 3 --       IIF(i, j, i', k, i)
                and not (k le 10)
            imp FW(i, j, i', k, i)

testfor2 # 4 --       IIF(i, j, i', k, i)
                and k le 10
                and 1 le k
            imp    IIF(i + j,
                      j, i', k + 1, i)
                and        not (k1 le 10)
                      and IIF(i2,
                              j,
                              i', k1, i + j)
                      and FW(i2,
                              j,
                              i', k1, i + j)
                   imp     IIF(i2,
                              j, i', k1, i)
                      and FW(i2,
                              j, i', k1, i)
```

IV.7. The Goto Statement

```
procedure testgoto (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
label 999;
begin
  i := 1;
  go to 999;
  i := 2;
999: assert B (i, j, i');
  i := 3
end;
```

There are 2 verification conditions for testgoto:

```
testgoto # 1 -- B(i, j, i') imp postc(3, j, i')


testgoto # 2 -- prec(i, j) imp B(1, j, i)
```

IV.8. The If Statement

```
procedure testif (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin
  if BB (i, j) then
    i := 1
  else
    i := 2
end;
```

There are 2 verification conditions for testif:

```
testif # 1 --    prec(i, j) and BB(i, j)
        imp postc(1, j, i)


testif # 2 --    prec(i, j) and not BB(i, j)
        imp postc(2, j, i)
```

IV.9. The Null Statement

```
procedure testnull (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin ; end;
```

There is 1 verification condition for testnull:

```
testnull # 1 -- prec(i, j) imp postc(i, j, i)
```

IV.10. The Procedure Call Statement

```
procedure testcall (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin
  testq (i, j)
end;
```

There is 1 verification condition for testcall:

```
testcall # 1 --      prec(i, j)
                 and computes(testq(i, j), result(i2))
               imp postc(i2, j, i)
```

IV.11. The Prove Statement

```
procedure testprove (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin
  prove B (i, j, i')
end;
```

There are 2 verification conditions for testprove:

```
testprove # 1 -- prec(i, j) imp B(i, j, i)


testprove # 2 --    prec(i, j) and B(i, j, i)
            imp postc(i, j, i)
```

IV.12. The While Statement (Inductive Assertions)

```
procedure testwhile1 (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin
  maintain I (i, j, i')
  while BB (i, j) do
    i := i + j
end;
```

There are 3 verification conditions for testwhile1:

```
testwhile1 # 1 --        prec(i, j)
            and I(i2, j, i)
            and not BB(i2, j)
          imp postc(i2, j, i)


testwhile1 # 2 -- prec(i, j) imp I(i, j, i)


testwhile1 # 3 --    I(i, j, i') and BB(i, j)
          imp I(i + j, j, i')
```

IV.13. The While Statement (Subgoal Assertions)

```
procedure testwhile_2 (var i: Integer; j: Integer);
pre prec (i, j);
post postc (i, j, i');
begin
  maintain IW (i, j, i', iInitial)
  while BB (i, j) do
    i := i + j
  thus CW (i, j, i', iInitial)
end;
```

There are 4 verification conditions for testwhile2:

```
testwhile2 # 1 --       prec(i, j)
                    and IW(i2, j, i, i)
                    and not BB(i2, j)
                    and CW(i2, j, i, i)
                 imp postc(i2, j, i)


testwhile2 # 2 --     prec(i, j)
                 imp IW(i, j, i, i)


testwhile2 # 3 --       IW(i, j, i', i)
                    and not BB(i, j)
                 imp CW(i, j, i', i)


testwhile2 # 4 --       IW(i, j, i', i)
                    and BB(i, j)
                 imp    IW(i + j, j, i', i)
                    and       not BB(i2, j)
                        and IW(i2,
                               j, i', i + j)
                        and CW(i2,
                               j, i', i + j)
                      imp     IW(i2, j, i', i)
                        and CW(i2, j, i', i)
```

Appendix V
Restrictions, Bugs, and "Curious Features"

The user is encouraged to study the message file <Affirm>KnownBugs.TXT from time to time; it contains the most up-to-date information on any problems the system is experiencing, as well as suggestions for avoiding or getting around these problems.

1. The upper and lower case letters "T" and "t" cannot be used as the name of any ***Affirm*** object.

2. Variables referenced on the left-hand-side of a rule definition may not be referenced on the right-hand-side. For example, the following is illegal:

```
schema induction(q) = = cases(Prop(NewSequenceOfElemType),
                              all q, i (IH(q) imp Prop(q apr i)));
```

3. ***Affirm*** output is not always acceptable as ***Affirm*** input. In particular, the system uses indentation to indicate precedence. It sometimes prints:

```
(a imp b) and c imp d
```

as

```
    a imp b
  and c
imp d
```

4. ***Affirm*** knows precious little about the properties of integers [§6.7].

5. Undoing commands out of order is a quick way to reach a state where the user can take the rest of the day off.

6. The print command lacks the ability to print most ***Affirm*** objects with any selectivity. It is either all or none.

7. Redeclaring variables, interfaces and discarding rules works, but old variables and old interfaces can appear in listing.

8. Very few of the ***Affirm*** objects have all three of the operations name, rename, and unname.

9. In expressions, operator precedence is not Pascal but is normal precedence.

10. Sections of the proof tree can automatically reappear [§7.8].

11. ***Affirm*** needs work on the scope model it uses. Only the variables of the current type are available while all the interfaces in the system are known. Variables mentioned in a particular theorem are tied to a particular type. You may get into trouble if you use a theorem created in a different type [§4.3].

12. This list is incomplete; see the Users Guide for invaluable advice, other confessions, and curiosities.

Appendix VI
Overviews of the Operating System Executives

This appendix contains brief summaries of the operating system executive commands available in the operating systems under which *Affirm* operates. (As of *Affirm* version 1.21, these are *Tops-20* and *Tenex*.) This summary is in no way intended to replace or substitute for the relevant manuals published by and available from the distributors of the operating systems.

Key:
⇋ = Escape
↲ = Carriage Return

VI.1. Tops-20 Executive Summary

Login
Log DirectoryName Password↲↲

Logout
Logo⇋↲

Directory
To see all the files in your directory:
Dir↲

Detach
Det⇋↲

Attach
Att DirectoryName↲ Password↲

Connect
To connect to another Directory:
Conn⇋ DirectoryName↲ Password↲

control-C
To return to Exec. This will abort a partially typed or partially executed command (sometimes you must type several control-C's).

control-T
To see the current load average of the system or the progress of a program. You may type this during most programs without stopping them.

control-F
This works like the Escape Key on file names, but only completes one part of the file name.

Type To see a file on your terminal screen:
Typ⇋ FileName⇋↲

Copy To copy a file from another directory on your machine:
Cop⇋ <DirectoryName>FileName⇋⇋↲

Rename
To rename a file:
Ren⇋ OldFileName⇋ NewFileName↲↲(no ⇋)

Print To send a file to the Lineprinter:
Print⇋ FileName⇋↵

Lineprinter and Penguin Queues
To see the lineprinter and penguin queues:
IQ↵

XPRESS
To send a file to the Penguin:
XPRESS FileName⇋↵

File Protection
To see if a file is protected:
Dir FileName⇋,↵
@@Pro↵↵ Filename;P777752 (This number indicates an unprotected file.) Filename;777704 or 775200 (Either of these numbers indicate a protected file.)

To change your file protection:
Set File Pro⇋ FileName⇋ 777752↵ (for unprotected file) and 777704↵ or 775200↵ (for protected file)

Change Password
To change your password:
Set⇋ Dir⇋ Pas⇋ DirectoryName↵ OldPassword↵ NewPassword↵ NewPassword↵

VI.2. Tenex Executive Summary

Login
Log DirectoryName Password↵↵

Logout
Logo⇋↵

Directory
To see all the files in your directory:
Di↵

Detach
Det⇋↵

Attach
Att DirectoryName Password↵

Connect
To connect to another Directory: Conn DirectoryName Password↵
To return to your Directory: Conn↵.

control-C
To return to Exec. This will abort a partially typed or partially executed command (sometimes you must type several control-C's).

control-T
To see the current load average of the system or the progress of a program. You may type this during most programs without stopping them.

control-F
This works like the Escape Key on file names, but only completes one part of the file name.

Type & TCopy
To see a file on your terminal screen:
Typ⇋ FileName⇋↵ or TC⇋ FileName⇋↵

Copy To copy a file from another directory:
Cop⇋ <DirectoryName>File Name⇋⇋↵

Rename
To rename a file:
Ren⇋ OldFileName⇋ NewFileName↵↵(no ⇋)

List and LCopy
To send a file to the Lineprinter:
List⇋ FileName⇋↵ (This method will give file information at the top of each page.)
LC⇋ FileName⇋↵ (This method will create a clean listing.)

LP To see the Lineprinter queue:
LP↵

XPRESS
To send a file to the Penguin:
COPY FileName⇋PNG:↵

PLP To see the Penguin print queue:
PLP↵

File Protection
To see if a file is protected:
Di FileName⇋,↵
@@Pro↵↵ Filename;P777752 (This number indicates an unprotected file.) Filename;777704 or 775200 (Either of these numbers indicate a protected file.)

To change your file protection:
Pro⇋ FileName⇋ 777752↵ (for unprotected file) and 777704↵ or 775200↵ (for protected file)

Change Password
To change your password:
Cha⇋ Pas⇋ DirectoryName OldPassword NewPassword NewPassword↵ (Notice that there are no < > around the directory name.)

Archives
To see your files in Archives, type:
Int⇋⇋↵
After the list of files is displayed, you will be asked if you want the most recent file retrieved. If you answer yes, you will get a message from the operator as soon as a copy of the file is in your directory. The original will remain in Archives.

To put a file in Archives, type:
Arch⇋ File⇋ FileName⇋↵

Appendix VII Glossary of Terms

axiom	As in mathematics, a statement or property accepted without further proof. In ***Affirm***, an axiom is an equation used to define the behavior of an abstract data type. Axioms are turned into automatically applied rewrite rules.
basis	1. The initial step of an induction proof. 2. The name of the abstract data type in ***Affirm*** in which the user is initially placed.
case analysis	A proof strategy which considers all the various possibilities separately; for example, $x \geq 0$ and $x < 0$ might be two cases.
definition	A rewrite rule which will not be expanded automatically, but rather only upon explicit user request. Definitions are often used for notation.
equation	Any expression of the form A = B where A and B are expressions of the same type.
event	Any command to ***Affirm***. Every event is assigned an event number, and most may be undone, redone, or fixed.
event number	The natural number associated with an event. The counter starts at one.
<u>some</u> list	A list of existentially quantified variables associated with a proposition. These may be instantiated in order to (attempt to) prove the associated proposition.
forall list	A list of universally quantified variables associated with a proposition. Also referred to as the <u>all</u> list.
history	A *window* of recent events, which remembers the command input and its side effects. Such events may be undone, redone, or fixed. As new events are added, old ones are forgotten.
infix	A binary operator which appears between its two operands rather than in front (prefix), e.g., a op b rather than op(a, b) ***Affirm*** uses the <u>infix</u> designation to format *output*; it accepts *input* in both prefix and infix forms for any (binary) operator, regardless of the output designation.
instantiation	The result of replacing an existentially quantified variable with a specific value.
proof forest	A directed acyclic graph which records the components of each theorem's proof. Theorems are roots in the forest.
proposition	A Boolean predicate to be proven. If the proposition contains any variables that are not explicitly quantified, they are assumed to be universally quantified, and are termed

	all or forall variables.
proven	The state reached when a proof exists of a lemma or theorem using only axioms, proved propositions, or explicitly assumed propositions.
rewrite rule	A rule of the form LHS → RHS, where both sides are expressions. Any part of a proposition which matches LHS will be *rewritten* to look like the RHS.
rewrite rule convergence algorithm	That process which seeks to guarantee that a set of rewrite rules has the unique termination property.
unique termination	A set of rewrite rules has this property if, given an expression, the expression always rewrites to the same final result, regardless of the order in which rules are applied.
finite termination	A set of rules has this property if there exists no finite expression which will cause the rules to forever rewrite. ***Affirm*** assumes, in seeking unique termination, that the user has demonstrated finite termination of the set of rewrite rules.
unit	The general term used to refer either to ***Pascal*** procedures, functions, or the main program.
variable	A symbol (such as x or s) which stands for some value of a data type.
verification condition	One of the logical formulae produced from a unit containing assertions by the verification condition generator. Verification conditions contain no programming constructs except expressions.

Appendix VIII
A Beginner's Subset of Affirm Commands

Not all commands are listed here. Rather, the *most useful* ones are enumerated, using the gross categories *Specification*, *System*, and *Theorem Prover*.

System

abort; Returns from a *lower executive*, aborting the pending command (see the ok command).

exec; Invokes the operating system as a lower fork.

fix; Places the text of a command in a text editor, and re-executes the revised command upon return from the editor.

freeze *fileName*;
Saves the entire state of the current system in file *fileName*.

gripe *file*;
Asks for the text of a message and then sends the message (using the Arpanet) to ISI.

load *file*;
Reads a file containing the internal form of a type specification previously saved using the save command.

needs type *typeNames*;
Causes the system to find and read the type specification for each specified type name.

note *comment*;
The comment facility.

ok; Returns from a *lower executive*, and then executes the pending command (see the abort command).

print *option furtherArguments*;
Prints something. Common *options* are:

theorems *names*;
Lists the indicated theorems.

prop *names*;
Lists the indicated propositions (they do not have to be theorems).

type *typeName*;
Lists the specification of the type.

file *fileName*;
Lists the file.

IH; Lists the current definition of the induction hypothesis IH.

known *objectName*;

Lists the names currently associated with elements of the indicated object class.

proof *theorems*;
Lists the proof trees of the indicated theorems.

profile; Enters a profile dialogue, where each profile entry is displayed and you have the option of providing a new value. The command can also take parameters: see the description in the command synopses.

quit; Closes the transcript file and returns to the operating system.

read *file*;
Reads a file of commands, executing each. Quite useful for reading the text form of type specifications.

readp *file*;
Reads a file containing Pascal programs, building the internal parsed form (see the genvcs command).

review; Puts the text of the transcript in a text editor, so you can retrace your steps.

save type *typeNames*;
Saves the internal form of the type specifications, each in its own file. The load command can be used to read the files.

undo *event*;
Undoes the effects of the indicated command.

Specification

adopt *typeName*;
Copies the declared variables from *typeName* into the current type. Useful for establishing proof contexts.

axiom *rule*;
Makes a rewrite rule L → R out of the rule L == R, and adds it to the system's rule set. All rules are applied to expressions during simplification, after each theorem-prover command.

declare *ids*: *typeName*;
Declares the names to be variables of the indicated type.

define *rule*;
Makes a rewrite rule out of the equation. Definitions are not automatically rewritten during simplification; you must explicitly request application using the invoke command.

edit *typeName*;
Opens a new type context for subsequent specification commands. See the type and end commands.

end; Ends the current type context and restores the previous one. Specification commands affect only the current type.

interface *op*(*params*): *typeName*;
Defines the domain and range information for an operation. *Params* are <u>variable</u> names, not <u>type</u> names.

rulelemma *rule*;
Treated just like an axiom.

schema *rule*;
Defines induction and case analysis schemas. See the description in the command synopses.

type *typeName*;
Establishes a new context for subsequent specification commands. If the *typeName* is already declared, it is totally redefined by this command. (But you can always <u>undo</u> this command!)

Theorem Prover

apply *name*, *expression*;
Applies the expression as a lemma to the proposition currently being proven.

cases; Raises embedded "Ifs" by applying the special rule

f(if b then x else y) → if b then f(x) else f(y)

employ *schemaName*;
Uses the schema to perform induction or case analysis.

eval *expression*;
Applies the normalization and simplification process to the expression.

genvcs *PascalUnitNames*;
Generates the verification conditions for the indicated Pascal procedures and functions.

invoke *definitionNames*;
Expands the references to a particular set of operations by replacing the reference with the definition.

name *newName*, *oldName*;
Name *oldName* to be *newName*. If *oldName* is omitted, then the proposition currently being proven is given the new name.

next; Moves to the next unproved part of the proof tree associated with the current theorem, in a fairly natural ordering.

normalize;
Simplifies the current proposition. It is not normally necessary to explicitly invoke this command, because it is automatically performed after each theorem-prover command.

put *existentialVar* = *expression*;
Instantiates existential quantifiers.

replace *expression*;
Performs equality substitutions.

search;
Attempts to find a set of instantiations of existential quantifiers that results in reducing the proposition currently being proven to true.

suppose *expression*;
Breaks the proposition *P* currently being proven into two propositions:

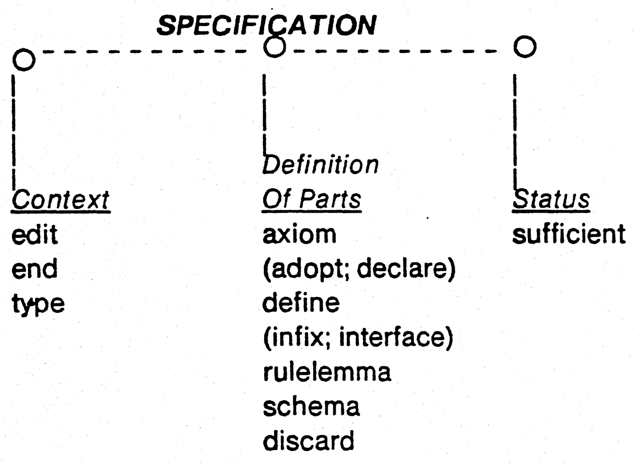
$expression \supset P$

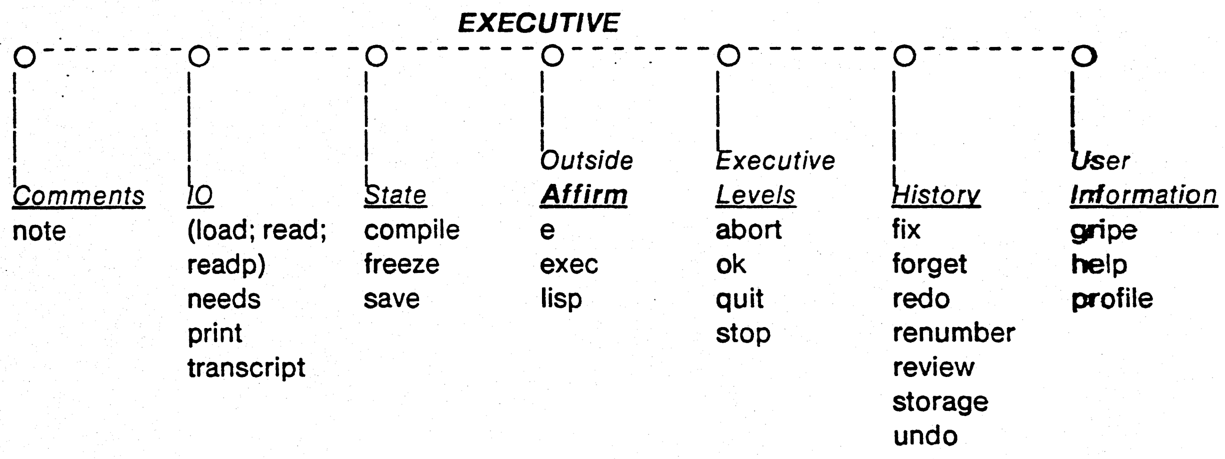
and

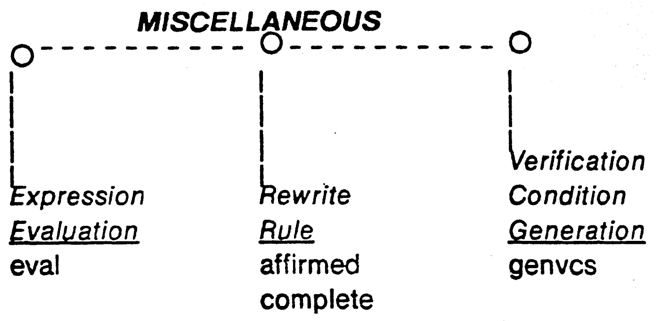
$expression \vee P$

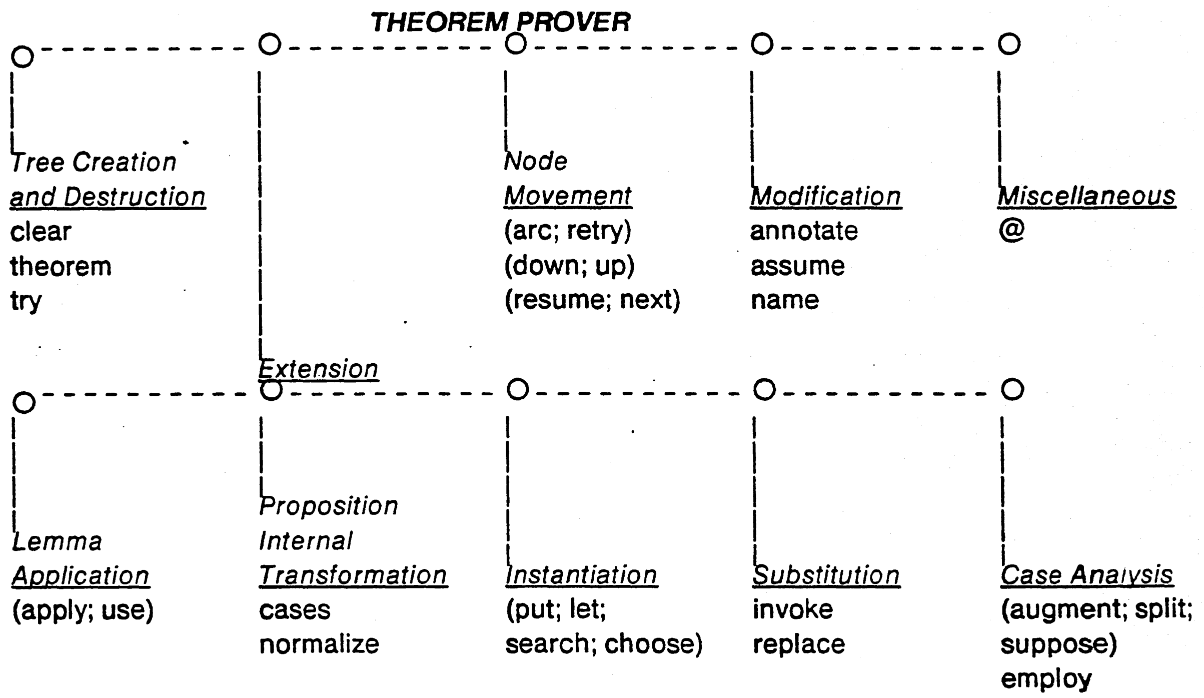
try *name*, *expression*;
Attempts the proof of the named expression.

Appendix IX
Command Structure Diagrams






MISCELLANEOUS
Expression Evaluation
eval
Rewrite Rule
affirmed
complete
Verification Condition Generation
genvcs


THEOREM PROVER
Tree Creation and Destruction
clear
theorem
try
Extension
Node Movement
(arc; retry)
(down; up)
(resume; next)
Modification
annotate
assume
name
Miscellaneous
@
Lemma Application
(apply; use)
Proposition Internal Transformation
cases
normalize
Instantiation
(put; let;
search; choose)
Substitution
invoke
replace
Case Analysis
(augment; split;
suppose)
employ

Appendix X
Command Synopses

This Appendix contains a synopsis of each ***Affirm*** command. For the most part, this synopsis is identical to the description given in the appropriate chapter of the reference manual. The descriptions are gathered here for convenience.

X.1. Affirm Commands

; Semicolon terminates most commands, except for subcommands in the @ (Interlisp editor) and e (escape-to-Interlisp) commands. It may also stand by itself, as a null command.

@ [*annotation*];

[§III.3] Places the user in the Interlisp editor, editing Current Proposition. The annotation is optional, and hopefully documents the less-than-mnemonic Interlisp editor commands that follow [§III.2], [§X.3].

abort;

[§3.5.4] Returns the user to the next higher ***Affirm*** executive (if there is one), and aborts the suspended command. The suspended command can then be fixed, or just forgotten.

adopt *typeName*;

[§4.2.1] Sometimes it is necessary to prove theorems about operators that are associated with types other than the current one. The operators of the type will be referenceable, because the type is in TypeSet. However, the variables of that type will not be referenceable in the current context. Rather than enter the necessary variable declarations manually, the adopt command provides a convenient way to *copy* all the declarations of a type over to the current one. Should any name conflicts occur, the variables being copied will be renamed by appending dollar sign characters ($) to them.

```
adopt SequenceOfElemType;
```

annotate [*proposition*,] *annotation*;

[§7.9.3] Attaches a comment to *proposition*; this will appear whenever *proposition* does. *Annotation* is arbitrary text, but cannot contain any semicolons. This is useful for

- documenting where and when an assumption was proven;
- noting what the user's plans are when the proof attempt returns to this spot; and
- commenting a tricky place in a proof.

apply [*nodeName*,] *proposition*;

[§7.4.1.1] This command adds *proposition* to Theorems, and adds it as a hypothesis to the Current Proposition. The command records this dependency by establishing the Uses relationship between the Current Proposition and *proposition*. *proposition* may be assigned a name. The expression corresponding to *proposition* will have its variables renamed to avoid conflicts with variables in the Current Proposition; the renamed form is printed on the terminal. The resultant Current Proposition is not printed,[37] since no meaningful simplification will occur

[37]But see the use command [§7.4.1.2].

until the user has performed instantiations. Typically, a put command will follow an apply command [§6.5].

arc *arcLabel*;

[§7.6.2.3] This command is used to move between cases. Somewhere above Current Proposition is a node with a child labelled *arcLabel*. That child becomes the new Current Proposition. For example, if an induction has three cases (emp:, apr:, and apl:), the user might wish to proceed in an unusual order, saying

```
arc apl:;
...
arc emp:;
...
arc apr:; .
```

assume [*nodeName*,] *proposition*;

[§7.9.1] Marks *proposition* as *assumed*: it is as if this node were proven (except that this special status is remembered). It may be given a name; this is useful if a file lists assumed facts (such as integer lemmas).

augment *proposition*;

[§7.4.2.1] *Proposition* is added as a hypothesis to the Current Proposition. Separately, the user must show that *proposition* can be deduced from the hypotheses already present. Any free variables in *proposition* are identified with those in the Current Proposition, rather than being renamed. Given a Current Proposition of the form

H imp C

this command spawns the two children:

- H imp *proposition*

- (H and *proposition*) imp C

These children are assigned the arc labels thesis: and main:, respectively.

axiom *rule* [, ..., *rule*];

[§4.2.3] each *rule* must be an equation *lhs* = = *exp*. The rewrite rule *lhs* → *exp* is (normally) added to RuleSet. Variables appearing in *exp* must appear in *lhs*. ***Affirm*** checks all proposed axioms to see how they affect the unique termination of RuleSet. It may interactively simplify the rule, reverse it, or add new rules [§5.2].

```
axioms LessLast(q apr x) = = q,
       LessLast(NewSequenceOfElemType) = = NewSequenceOfElemType,
       Last(q apr x) = = x;
```

cases ;

[§6.3.3] Distributes functions over ifs in the Current Proposition.

choose *path*;

[§6.6] Related to the search command, this command allows the user to pick some sequence of instantiations tried by the search command. The search command prints a small integer label to the left of each instantiation it attempts. The sequence of numbers describing the choice--*path*--is the parameter to the choose command. This command is useful if search found lengthy instantiations, but was unable to achieve a final proof.

clear proof;

[§7.10.1] Empties the Proof Forest and Theorems. Erases all proposition names, annotations, and assumptions. Fortunately, this command is undo-able.

compile type *typeName* [, ..., *typeName*];

[§9.6] Writes a file containing a compiled version of the internal representation of a data type specification (Interlisp code). All stable types should be compiled, since this form of the type uses the least space and runs the fastest. Any types still undergoing development should be saved, rather than compiled.

complete ;

[§5.4] Attempts to prove the Current Proposition by *reductio ad absurdum* (proof by contradiction). It does this by negating the conclusion of Current Proposition, forming a rewrite rule from it, and (temporarily) adding it to RuleSet. Each hypothesis of Current Proposition is also turned into a rewrite rule and (temporarily) added to RuleSet. The algorithm then tries to generate a contradiction in RuleSet, by performing the unique termination test. If the rule

true → false

is generated, the Current Proposition is proved by contradiction. Otherwise, the final set of rules is used to construct a new result, which may be somewhat simpler than the Current Proposition. This command is sometimes used specifically to rearrange the clauses of a proposition in an inconvenient form.

declare *id* [, ..., *id*]: *typeName*;

[§4.2.1] Each *id* is declared to be a variable of type *typeName*. *typeName* must be a member of TypeSet. Each of the declarations is added to the local declaration set of the current type.

```
declare q, q1: SequenceOfElemType;
declare x: ElemType;
```

define *rule* [, ..., *rule*];

[§4.2.3] each *rule* is an equation *lhs* = = *exp*. Definitions are rewrite rules, but these rules are only applied when specifically invoked by the user with the invoke command [§7.5.2.3]. Definitions are generally used to simplify notation: they are only invoked when needed, so that their contents do not overly complicate propositions. Variables in *exp* must either be bound quantifiers or must appear in the corresponding *lhs*, *but not both*.

discard disconnected;

[§7.10.3] Any nodes which are disconnected (not part of the proof tree of any theorem) are destroyed. Their expressions, annotations, names, and proofs go away. This can save a considerable amount of space. Since the command is undoable, space is only reclaimed when this event is forgotten [§3.4].

discard history;

[§3.4] Purges the history window. This command can be undone.

discard interface *interfaceName* [, ..., *interfaceName*];

[§4.2.2] Discards the indicated operations in the current type. Each operator must be defined in the current type. Note that any references to the discarded operations are inconsistent. The system does not check for this condition. It is the user's responsibility to discard or redefine any rules or propositions referencing the newly-discarded operations.

discard lhs *lhs*;

[§4.2.3] *Lhs* must be the left hand side of some axiom, rulelemma, definition, or schema. The rule in <u>RuleSet</u> with left hand side identical to *lhs* is removed from <u>RuleSet</u>. This may destroy the unique termination of <u>RuleSet</u>; no check for this condition is performed.

discard theorem *nodeName* [, ..., *nodeName*];

[§7.10.2] Removes the designated nodes from <u>Theorems</u>. It thus no longer has a proof state, and disappears from summaries of theorems. This is useful when an incorrect lemma has been stated. It is not permissible to discard a lemma which is applied in the proof of some other theorem. If one of the $nodeName_i$ applies another as a lemma, that is okay. ***Affirm*** sorts the list first, and removes the *uses* relationship when the using theorem is deleted. These nodes continue to exist, and retain their proofs; they form part of the disconnected nodes in the tree. The <u>try</u> command will reverse the effects of <u>discard theorem</u>.

discard variable *variableName* [, ..., *variableName*];

[§4.2.1] Discards the indicated variables from the current type. Note that any <u>use</u> of the variables, such as in interface declarations or rules, is now undefined, and may be <u>inconsistent</u>. The system does not presently check for this condition. However, the user will certainly feel the effects later! It is the user's responsibility to discard or redefine interfaces and rules referencing the newly-discarded variables.

denote *expression* by *variableName* [, ..., *expression* by *variableName*];

[§7.5.2.5] For each *expression-variableName* pair, this command replaces all occurrences of *expression* with *variableName*, and adds to the <u>Current Proposition</u> the hypothesis

expression = variableName

down [*child*];

[§7.6.2.2] The <u>Current Proposition</u> must have children; this command descends to one of them. *Child* may be:

- an arc label;
- the name of a child;
- an ordinal number (between 1 and the number of children of <u>Current Proposition</u>);
- a node number (if *child* > # children) (*This option is not particularly recommended*); and
- omitted: the first untried child is picked. That failing, the first child is picked.

e *InterlispCommand*

[§III.1] Note that the customary semicolon does <u>not</u> terminate this command. The Interlisp interpreter is invoked with the one command; after the interpreter prints its result, the user is returned to the ***Affirm*** executive.

edit *typeName*;

[§4.2] *TypeName* must be a member of <u>TypeSet</u>. *typeName* is pushed onto <u>ContextStack</u>, thus making the local declarations of *typeName* available for referencing.

employ *schemaName*(*var*);

[§7.4.2.3] This command permits the use of induction, using any induction schema defined in the relevant abstract data type. *SchemaName* must have been defined (using the <u>schema</u> command) for objects of the same data type as *var*. If the right hand side of the schema definition is of the form

```
cases(C1, ..., Cn)
```

then the resulting propositions $\{C_i\}$ are set up as children of the Current Proposition. The $\{C_i\}$ are expressed in terms of the special predicates Prop(x) and IH(x) which are defined as though the commands

```
axiom Prop(var) = = Current Proposition;
define IH(var)  = = Current Proposition;
```

had been given. For example, suppose that Current Proposition, named *SubExtends*, is

```
sub(q, q1) imp sub(q, q1 apr x)
```

If the command

```
employ induce(q);
```

is performed in the context of

```
schema induce(q) = = cases(Prop(emp),
                          all q0, x0 (IH(q0) imp Prop(q0 apr x0)));
```

the following children would result (before simplification):

```
1.  sub(emp, q1) imp sub(emp, q1 apr x)

2.              IH(q0, 2 {SubExtends[38]})
            and sub(q0 apr x0, q1)
    imp sub(q0 apr x0, q1 apr x)
```

Quite often, the simplest cases, such as (1), above, normalize to true; the user is notified of these instances. The cursor will automatically be moved to the first nontrivial child.

The cases of an induction are given system-generated labels; these derive from the primary operators underneath Prop in the schema. For example, the above sample would have labels emp: and apr:. Induction is subject to the soundness constraint that *var* must be contained in the all list, and may have no Skolem dependencies upon any variables in the some list [§6.2.2].

end ; [§4.2] Causes ContextStack to be popped, ending the current type's specification and returning to the previous context.

eval *expression*;
[§6.4] Simplifies *expression*, and prints the result. This is useful for testing and demonstrating abstract data types.[39] For more details on its use, see the Users Guide.

exec ;[§3.10.6] Invokes the operating system executive as a subroutine. The user can do anything that can be done at the original executive without destroying the files and memory associated with ***Affirm***. To continue with the ***Affirm*** session, the user should type POP at the operating system executive command level.

fix [*eventNumber*];
[§3.4] Places the user in a text editor (determined by the profile entry *TextEditor*) with the text of the command issued at event *eventNumber*. The default event when *eventNumber* is not

[38] The two-parameter reference to IH is explained on page 55 [§7.4.2.3].

[39] The user profile entry *ShowRules* [§3.13] is useful for observing the application of axioms to sample expressions.

explicitly supplied is the previous event.

freeze [*fileName*];
[§9.6] Causes the entire system state[40] to be written into file *fileName*. The default freeze file name when none is provided in the command is determined by the user profile entry *FreezeFileName*. The size of the file written is on the order of 300 pages. This file can then be run at a later time by simply typing the file name at the operating system executive level. The user will then be back in ***Affirm*** at the executive, as if the freeze had never happened (except that a new transcript file will be opened, if necessary). This command is quite useful for freezing a session in place, and then continuing it later. Compare this with the save command, which does not save the entire system, but just relatively small components of it.

genvcs *procedureName* [, ..., *procedureName*];
[§8.3] Each *procedureName* must be a Pascal procedure or function[41] unit previously parsed via the readp command. Verification conditions are generated for each *procedureName*.

gripe *subject*;
[§3.8] Creates a message to be sent via the ARPANET to ***Affirm*** maintenance personnel. The system will ask the user to type the body of the message, which is terminated by **control-Z**. After the message is completed, the user has the options of sending the message, or aborting the gripe. The transcript [§9.2] can also be sent along as a separate message if it is pertinent to the documentation of the problem or suggestion.

infix *operatorName* [, ..., *operatorName*];
[§4.2.2] Each *operatorName* is declared to be an infix operator.

interface *expression* [, ..., *expression*]: *typeName*;
[§4.2.2] Just as declare establishes the types of variables, interface provides the necessary characteristics of operators. All operators should be declared using the interface command before they are referenced in other ***Affirm*** commands. Each *expression* will be an expression of the form *operatorName*(var_1, ..., var_m), where each of the var_i is a variable declared in the current type. The interface declaration states that *operatorName* is a function of *m* arguments, with argument types corresponding to those of the var_i. The value returned by *operatorName* will be of type *typeName*. In the case of an operator taking no arguments, the parentheses may be omitted. *infix* notation, such as q apr x, can also be used.

```
interface q apr x, apl(q, x): SequenceOfElemType;
```

invoke *rangedOp* [, ..., *rangedOp*];
[§7.5.2.3] Each of the specified operators should occur in the Current Proposition and have a definition [§4.2.3]. The definition is expanded. If an operator appears in its own definition, the new occurrence will not be expanded; thus the process will not loop. An ordinal range may be specified; if it is not, the first occurrence of each operator will be expanded. Some examples:

---

[40] The freeze command does not save the state of any open files.

[41] As of ***Affirm*** version 1.21 the verification condition generator did not process functions correctly.

invoke IH;	invoke the first IH
invoke IH |2|;	second IH
invoke IH |all|;	all IH's
invoke IH |-2|;	next to last
invoke IH |2:4|;	second, third and fourth
invoke F(i,j)|2:5|, G|3,5|;	second through fifth occurrences of F(i,j) and the third and fifth occurrences of G

This command can be automatically invoked by the *AutoInvokeIH* profile entry [§3.13].

let *var* = *exp* [, ..., *var* = *exp*];

[§6.5] This command has the same effect as the put command, except that the new result is the *disjunction* of the unchanged and the instantiated versions of Current Proposition. Thus, all variables in the some list remain subject to further instantiation with the put or let commands. This is useful if the user is not quite sure about an instantiation, or wishes to perform multiple instantiations. It does, however, double the size of the expression. If, for example, the Current Proposition was

```
all x some y(x): P(x, y)
```

The command

```
put y = x;
```

would give (before simplification)

```
all x: P(x, x)
```

while the command

```
let y = x;
```

would yield (before simplification)

```
all x some y(x): P(x, y) or P(x, x)
```

lisp ; [§III.1] The ***Interlisp*** interpreter is invoked. The user can next perform any Interlisp command. The OK command (without a semicolon after it) returns the user to the ***Affirm*** executive.

load [*fileName*];

[§9.6] Causes ***Affirm*** to load file *fileName*. The file must have been previously written using the save command. A data type specification is the only ***Affirm*** object that can be saved and then loaded. Note that the file's contents are *not* normal text, and cannot be directly modified by the user.

name *nodeName*, [*proposition*];

[§7.9.2] Christens the proposition; the system will henceforth refer to it by the name *nodeName*. If this name is already in use, the system displays its old value.

needs type[s] *typeName* [, ..., *typeName*];

[§9.5] Should be used immediately after a type command, before any other part of the type specification. This command ensures that each *typeName* are either loaded or read, before any more of the specification of the current type is processed. If the type is already defined, no further processing occurs. If it is not yet defined, then the most recent version of its specification is found. The algorithm which finds the files containing the types to be defined searches a set of directories for the most recent version of the specification of each type, whether that version be in original source form or in the internal saved form, or even in compiled form. For each type requiring such a directory search, ***Affirm*** first identifies the possible set of files containing versions of the type specification; it then ranks the versions (by using the file write date to determine which file was most recently written). ***Affirm*** will normally then proceed to load or read that file, as is appropriate, unless the profile entry *TypeNeeds* is set to Ask. The user will be asked to point out the correct type specification to be input. The set of directories used as of ***Affirm*** version 1.21 is {*Connected*, *Login*, PVLibrary, Affirm}.

next ; [§7.6.1.4] Moves to the next task, according to a depth-first plan, using the following hierarchy:

1. If the Current Theorem has leaves, move to the next one, in a left-to-right ordering of the leaves of the proof tree.

2. If the Current Theorem uses an unproven lemma, try it.

3. If the Current Theorem is used as a lemma by an unproven theorem, return to the theorem. This process extends to any unproven ancestor.

4. If none of the above hold, then stay put and perform the command

   print unproven;

Within this hierarchy, the most-recently-attempted theorem is preferred. Where possible, resume.

normalize ;
[§6.3.2] Causes the Current Proposition to be (again) normalized and printed. Since propositions are normalized upon becoming the Current Proposition, this will normally have no effect, but may be necessary due to the occasional incompleteness of the simplification process.

note *arbitraryTextExceptSemicolon*;
[§3.10.1] The text is placed in the transcript. No other processing is performed.

ok ; [§3.5.4] Returns the user to the next higher ***Affirm*** executive (if there is one), and resumes processing of the suspended command. If this command still has errors in it, the user may well be placed into a lower executive once again. The abort command is useful here, too.

print ?;
[§7.7.1] print ?;
displays a list of all the keywords that can follow the command word print. Equivalent to the command

    print known PrintObjects;

print assumptions;
[§7.7.1] Lists all the assumed propositions, and the theorems that depend on them.

print BadEquations;
[§7.7.1] Lists the rules that have been suppressed during the various Knuth-Bendix [§5.2] convergence tests, if any.

print both [list | nolist] *whatNodes*;
[§7.7.1] Like print proof but lists all the propositions in the proof tree. *Verbose.*

print file *fileName*;
[§7.7.1] Copies the contents of file *fileName* to the terminal, and also to the transcript.

print history;

[§7.7.1] Prints the user-issued commands still resident in the history window.

print IH;

[§7.7.1] Prints the definition of each of the inductive hypotheses [§7.4.2.3] in the Current Proposition (if any).

print known *objectName*;

[§7.7.1] Enumerates the currently defined set of names in the object class *objectName*. The object names as of ***Affirm*** version 1.21 are AffirmObjects, Arcs, Axioms, Commands, Definitions, Directories, Files, FileTypes, Interfaces, Lemmas, Nodes, PrintObjects, ProfileEntries, Schemas, TypeParts, Types, and Variables.

print [parts *typeParts*] [types *typeName* [, ..., *typeName*]] lhs *lhs* [, ..., *lhs*];

[§7.7.1] This command provides the rudimentary capability of listing those rules that match some *pattern*. *TypeParts* is a list selected from the set {axiom, lemma, defn, schema}; the list may be empty, in which case the default value axiom is used. *TypeNames* is a list of type names; the list may be empty, in which case the keyword need not be typed. The default value is the list of all currently defined types. *Pattern* is an expression, restricted to one of two simple forms: operator, or operator1(operator2). Each rule in the requested set of types that is a member of one of the requested parts is pattern-matched against the pattern; if it succeeds, the rule is listed. If it fails, the rule is ignored. Only the left-hand side of a rule is used in the pattern-matching process. If the pattern is a simple operator, a match succeeds if the *main* operator of the left-hand side is this operator. If the pattern is of the form operator1(operator2), then operator1 is the main operator, and operator2 is *any internal* operator. If the left-hand side of a rule has operator1 as its main operator, and contains a reference to operator2 as an internal operator, the match succeeds. For example, the command

```
print lhs join(apr);
```

will list all the axioms whose left-hand-side main operator is *join*, and which also reference the operator *apr* as an internal operator. This example is useful for the type SequenceOf*X*, for quite a few types *X*.

print next;

[§7.7.1] This command displays the proposition that the next command would make the Current Proposition.

print original;

[§7.7.1] Prints the unnormalized form of the Current Proposition (not particularly useful).

print proof [list | nolist] *whatNodes*;

[§7.7.1] Displays the proof tree. The default for *whatNodes* is `T`. `List` causes any lemmas that are used in the proof of Current Theorem to be listed. Note that *whatNodes* does not have to be a theorem, so the user can print a partial proof tree.

print prop [list | nolist] *whatNodes*;

[§7.7.1] Lists the propositions and their associated names. For example,

```
print prop T;
```

prints Current Theorem.

print result;

[§7.7.1] Prints the Current Proposition [§7.2] in its normalized form.

print status [list | nolist] [*whatNodes*];
[§7.7.1] Tells whether the specified theorems are tried, untried, awaiting lemmas, proved, or assumed. The default when *whatNodes* is omitted is `theorems`.

print type *typeName*;
[§7.7.1] *TypeName* must be a member of TypeSet. The declarations, needs, interfaces, infix operators, axioms, rulelemmas, definitions, and schemas of type *typeName* are printed on the terminal. Should only a subset of these be desired, *typeName* may be followed with a list of *qualifiers*.

```
print type ElemType;
print type SequenceOfElemType decl schema;
```

print unproven;
[§7.7.1] Prints the status of all unproven theorems.

print uses [*whatNodes*];
[§7.7.1] Which lemmas are used where? The default for *whatNodes* is `theorems`.

print variables;
[§7.7.1] Lists just the *variables* in the Current Proposition; this is useful if the expression is too big to be conveniently displayed as a whole.

profile ;
[§3.11] The profile enquiry dialogue is initiated with the user. The question mark command ? is quite useful here in order to determine what the options are at each step.

profile *profileEntryName* [= *value*] [, ..., *profileEntryName* [= *value*]];
[§3.11] Each referenced profile entry is either displayed with its current value or modified, as is appropriate.

put *var* = *exp* [, ..., *var* = *exp*];
[§6.5] Each *var* must be a variable in the some list of the Current Proposition. Each *exp* is an expression upon which the corresponding *var* can legally depend [§6.2.2]. The *exp* is substituted for the corresponding *var*.

quit ; [§3.10.6] Stops ***Affirm***, returning to the operating system executive. The user can return to ***Affirm*** by typing CONTINUE at the operating system executive command level.

read [*fileName*];
[§9.6] Causes ***Affirm*** to read *fileName*. The file must contain ***Affirm*** commands. The last command in the file must be the stop command. *FileName* is a text file that the user presumably created using some text editor.

readp [*fileName*];
[§8.2] Causes ***Affirm*** to read *fileName*. The file must contain Pascal programs. *FileName* is assumed to be a text file.

redo [*eventNumber*];
[§3.4] Re-executes the command at event *eventNumber*.

replace [*expression* [, ..., *expression*]];
[§7.5.2.2] If no argument is given, then every hypothesis in Current Proposition of the form L = R is used to replace all other occurrences of L with R. Each *expression* should occur in an equality hypothesis (of the form *expression* = *R* or *R* = *expression*). All other occurrences of *expression* are replaced with *R*. For example, if Current Proposition is

(fee(j, k) and j = m and n = k) imp fie(m, n)

replace; will yield

(fee(***m***, k) and j = m and n = k) imp fie(m, ***k***)

while the command replace m, n; will yield

(fee(j, k) and j = m and n = k) imp fie(***j***, ***k***)

resume ;

[§7.6.1.2] The Current Theorem must be tried. The Current Proposition is restored to the value it had when the user was last proving this theorem, thus resuming a partially-completed proof. The resume command is usually preceded by a try command.

retry ; [§7.6.1.3] This command is equivalent to the (otherwise unspeakable) command

try Current Theorem;

In other words, this command retries the current theorem.

review ;

[§3.10.3] Places the user in a text editor determined by the profile entry *TextEditor*, with the transcript file. The user can then use editor commands to review the events in the file. Each command begins with "U:".

rulelemma *rule* [, ..., *rule*];

[§4.2.3] The rulelemma command is a synonym for the axiom command.

save type *typeName* [, ..., *typeName*];

[§9.6] Causes ***Affirm*** to write files containing the specifications of the indicated types. The file name of each file is the upper-case version of the corresponding type name. The save command can be used in conjunction with the load command to remember data type specifications across ***Affirm*** sessions. The file written by the save command for each type is the internal form of the type specification (Interlisp code). Thus little processing is required to load the type back into ***Affirm***, compared to the processing required when first creating the specification. The file name of the file is obtained by upper-casing the type name; thus type names may not differ only in casing, due to the possible file name conflict.

schema *rule* [, ..., *rule*];

[§4.2.3] Each *rule* is an equation *lhs* = = *exp*. The schema command introduces induction rules. The soundness of schemas is not determined by ***Affirm***; the user must establish this property. It is in schema declarations that the restriction imposed on equations is most often felt. The following declaration illustrates a very common error:

```
schema Induction(q) = =
          cases(Prop(NewSequenceOfElemType),
                all q, x(IH(q) imp Prop(q apr x)));                    (bad!)
```

Here the parameter is the same identifier as the quantifier in the expression. A correct schema declaration would be:

```
schema Induction(q) = =
          cases(Prop(NewSequenceOfElemType),
                all q0, x(IH(q0) imp Prop(q0 apr x)));
```

search ;

[§6.6] Uses the method of *chaining and narrowing* [§6.6] to attempt to automatically find the instantiations sufficient to reduce Current Proposition to true. The command displays the sets of instantiations it tries. These may be referenced by the user in the choose command.

set *variable* to *expression*;
[§6.4] *Variable* no longer represents itself; it is assigned a value which will replace it whenever an expression is normalized. This effect is permanent until *variable* is explicitly given another value. This may be useful in conjunction with the eval command. Other than that, it is not recommended.

stop ; [§3.5.4] Should be used only in a file of ***Affirm*** commands, as the last command. It avoids the usual end-of-file problems.

storage *degree*;
[§3.4] *Degree* is one of {normal, severe, tight}.

sufficient? [*typeName*];
[§4.4] *TypeName* must be a member of TypeSet. A sufficient-completeness check is performed and the results displayed on the terminal.

suppose [*proposition*];
[§7.4.2.2] This command splits the Current Proposition into two children:

- *proposition* imp Current Proposition

- *proposition* or Current Proposition

These children are labelled yes: and no:. If *proposition* is not supplied, the splitting predicate is automatically generated by ***Affirm*** using the internal If-Then-Else form of Current Proposition. Basically, the predicate is chosen from the first significant branch point. For example, if the Current Proposition is of the form

((A imp B) and H) imp C

the suppose command will yield the two children

A and B and H imp C *and* (~A) and H imp C

the children generated by the suppose command when no explicit proposition is supplied are labelled first:, second:, etc. It usually produces only two. Its detailed description follows, but it is usually best to experiment.

If Current Proposition is of the form:	The children are:
if B then C1 else C2	{B imp C1, B or C2}
C_1 and C_2 and ... and C_k	{C_k, C_1 and ... and C_{k-1}}
H imp (C_1 and ... and C_k)	{H imp C_1, (H and C_1) imp C_2, (H and C_1 and C_2) imp C_3, ..., (H and C_1 and ... and C_{k-1}) imp C_k}
(H1 and (H2 imp C1) and H3) imp C2	{(H1 and H2 and C1 and H3) imp C2, (H1 and (~H2) and H3) imp C2}

split ; [§7.4.2.2] A synonym for the suppose command with no parameter.[42]

swap *rangedExp* [, ..., *rangedExp*];
[§7.5.2.4] This command reverses equality hypotheses in the Current Proposition. Thus, it is often useful in conjunction with the replace command [§7.5.2.2]. Each *rangedExp* specifies one or more equalities to be reversed. Such a specification may give one of the arguments to the equality, or an ordinal range, or both. For example:

swap a;	Swap all equations whose left hand side (or right hand side) is the expression *a*.
swap |2|, |-2|;	Swap the second equation, and the next-to-last equation.
swap a |-1|;	Swap the last equation whose left-hand or right-hand side is the expression *a*.

thaw [*fileName*];
[§3.10.4] This command is the opposite of the freeze command. It takes one parameter, the name of a file containing a session frozen by a freeze command. Most users will not ever have a use for this command, since the frozen session can be started in ***TOPS-20*** or ***Tenex*** simply by typing the file name at the operating system executive level.

theorem [*nodeName*,] *proposition*;
[§7.3] This command simply enters the proposition into Theorems. It does not affect Current Proposition or Current Theorem. The command creates a root in the Proof Forest that may later be attempted. The user may associate a name with the theorem. This command is especially useful for command files containing lists of theorems to be attempted.

transcript [*fileName*];
[§9.2] Begins a (new) transcript file *fileName*. If there is no transcript file at the time the user issues this command, then the file name of the new transcript, if not provided in the command, is governed by the profile entry *TranscriptFileName*. If there is a transcript file at the time this command is issued, then the new file name, if not provided in the command, is identical to the old file name, with a new version number. The transcript file when the system first begins is written into the user's login directory, rather than the connected directory. Later transcript commands default to the connected directory.

[42]The split command is an obsolete command; its function has been merged into the suppose command.

transcript *toggle*;

[§9.2] *Toggle* is one of {off, on}. This command turns transcript processing either off or on. The file name is determined from the profile entry *TranscriptFileName*.

try [*nodeName*,] *proposition*;

[§7.6.1.1] Makes *proposition* be the Current Proposition. If *proposition* is in Theorems, it becomes the Current Theorem; otherwise, this designation is applied to its parent theorem. If *proposition* is new, it is added to Theorems. *proposition* is normalized and printed. This command is used for

- random access in a proof tree; and
- starting or resuming a proof (but see the description of the resume command [§7.6.1.2]).

type *typeName*;

[§4.2] Specifies *typeName* as the name of an abstract type, whose specification will be given by subsequent commands. The name *typeName* is added to the TypeSet and is pushed onto ContextStack. If *typeName* is already a member of the TypeSet, its existing specification *will be discarded*. Each new type is automatically provided with one variable declaration (the name of which is controlled by the profile entry *DummyVarName*), a declaration of an equality operation, and an axiom explicitly stating that the equality operation is reflexive. The remaining properties of an equality operation are *assumed*, and should be validated by the creator of the type.

undo [*eventNumber*];

[§3.4] Undoes the effects of execution of the command at event *eventNumber*, if possible.

up [*integer*];

[§7.6.2.1] Moves the cursor up to a predecessor in the tree. If the Current Proposition is already a theorem, this command has no effect. The number of ascensions defaults to 1.

use [*nodeName*,] *proposition*;

[§7.4.1.2] This command is exactly like the apply command, but also prints the new Current Proposition.

X.2. Interlisp Commands: Useful Interpreter Commands

DA	Prints the time of day. This command is also an operating system executive command.
EXEC	Invokes the operating system executive as a subroutine. The user should type POP to return to Interlisp.

X.3. Interlisp Commands: Useful Editor Commands

These commands can be used only in the Interlisp editor as subcommands of the @ command [§III.2], [§III.3].

!0	Modifies the focus of attention to be the parent of the current expression.
↑	Resets the focus of attention to the entire initial expression.
↑ 5 2	Modifies the focus of attention to be the sequence of hypotheses of Current Proposition.

hyp Same as ↑ 5 2.

con Same as ↑ 5 3.

↑ 5 3 Modifies the focus of attention to the conclusions of Current Proposition.

n *n* is a positive integer. This command moves the focus of attention to the *n*th element of the current expression. Caution: the command (*n*) deletes the *n*th element.

BK Modifies the current expression to be the previous sibling if possible.

(delete *n*) The *n*th element of the current expression is deleted.

(delete n_1 n_2 n_3)
The children at the listed positions are deleted. These indices are instantaneous, not one-at-a-time.

eval The current expression is evaluated.

(extract *n*)
The current expression is replaced with its *n*th child. For example, if the current expression is `(AND e1 e2 e3)` then

The command:	will result in:
(delete 2)	`(AND e2 e3)`
(delete 2 3)	`(AND e3)`
(extract 2)	`e1`

It is not sound to delete operators.

F *pattern* The F command attempts to find *pattern* within the current expression. If this search is successful then the focus of attention becomes the expression that matches *pattern*. *Pattern* can be any atom, and can contain escapes (which the operating system indicates as $). Each escape can match zero or more contiguous characters in an atom, e.g., VER$ matches VERYLONGATOM. The command will print a message if it cannot find the pattern.

infix The current expression is printed in infix form.

(invoke *definedName*)
The first instance of the definition with name *definedName* in the current expression is expanded.

NX This command moves the focus of attention to the next sibling. For example, if the expression being edited is

(PLUS (FOO 2) (FUM 3))

and the current expression is

(FOO 2)

then the NX command would focus upon

(FUM 3)

This command is very useful after the user uses the *n* command and then discovers that he or she mis-counted.

ok The user is returned to the ***Affirm*** executive, and the modified expression becomes Current

Proposition.

Pa	This command prints the current expression, showing the structure, (but not the contents) of contained subexpressions, a few levels deep.
PPa	This command pretty-prints the current expression.
stop	The edit is aborted; no changes are made to Current Proposition, and the user is returned to the ***Affirm*** executive.

Bibliography

[Dijkstra 76] Dijkstra, E. W., *A Discipline of Programming,* Prentice-Hall, 1976.

[Floyd 67] Floyd, R. W., "Assigning meanings to programs," in J. T. Schwartz (ed.), *Proceedings of Symposia in Applied Mathematics*, pp. 19-32, American Mathematical Society, 1967.

[Good 75] Good, D. I., R. L. London, and W. W. Bledsoe, "An interactive program verification system," *IEEE Transactions On Software Engineering* SE-1, (1), 1975, 59-67.

[Guttag 75] Guttag, J. V., *The Specification and Application to Programming of Abstract Data Types*, Ph.D. thesis, University of Toronto, Department of Computer Science, October 1975.

[Guttag 78a] Guttag, J. V., and J. J. Horning, "The algebraic specification of abstract data types," *Acta Informatica* 10, 1978, 27-52.

[Guttag 78b] Guttag, J. V., E. Horowitz, and D. R. Musser, "The design of data type specifications," in R. T. Yeh (ed.), *Current Trends in Programming Methodology*, pp. 60-79, Prentice-Hall, 1978. (An expanded version of a paper which appeared in *Proceedings of the Second International Conference on Software Engineering*, October 1976.)

[Guttag 78c] Guttag, J. V., E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *CACM* 21, December 1978, 1048-1064. (Also USC/Information Sciences Institute RR-76-48, August 1976.)

[Huet 78] Huet, G., *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, IRIA - LABORIA, Technical Report LABORIA Report No. 250, 1978.

[Jensen 75] Jensen, K., and N. Wirth, ***Pascal*** *User Manual and Report,* Springer-Verlag, New York, 1975.

[Knuth 70] Knuth, D. E., and P. B. Bendix, "Simple word problems in universal algebras," in J. Leech (ed.), *Computational Problems in Abstract Algebra*, pp. 263-297, Pergamon Press, New York, 1970.

[Lampson 77] Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report on the programming language Euclid," *SIGPLAN Notices* 12, (2), February 1977, 1-79.

[Lankford 78] Lankford, D. S., and D. R. Musser, On Semi-deciding First-Order Validity and Invalidity, 1978. (unpublished manuscript)

[London 78] London, R. L., J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek, "Proof rules for the programming language Euclid," *Acta Informatica* 10, (1), January 1978, 1-26.

[McCarthy 63] McCarthy, J., "A basis for a mathematical theory of computation," in Braffort and Hirschberg (ed.), *Computer Programming and Formal Systems*, pp. 33-70, North-Holland, 1963.

[Morris 77] Morris, J. H., Jr., and B. Wegbreit, "Subgoal induction," *Communications of the ACM* 20, (4), April 1977, 209-222.

[Musser 77] Musser, D. R., "A data type verification system based on rewrite rules," in *Proceedings of the Sixth Texas Conference on Computing Systems*, pp. 1A22-1A31, Austin, Texas, November 1977.

[Musser 80] Musser, D. R., "Abstract data type specification in the ***Affirm*** system," *IEEE Transactions on Software Engineering* SE-6, (1), January 1980, 24-32.

[Pratt 78] Pratt, V. R., Two Easy Theories whose Combination is Hard, 1978. (unpublished memo)

[Robinson 65] Robinson, J. A., "A machine-oriented logic based on the resolution principle," *Journal of the ACM* 12, (1), January 1965, 23-41.

[Teitelman 78] Teitelman, W., *Interlisp Reference Manual,* Xerox Palo Alto Research Center, Palo Alto, California, 1978.

[Wile 79] Wile, D. S., POPART: Producer of Parsers and Related Tools, 1979. (USC/Information Sciences Institute Technical Report, in preparation.)

Index