An Overview of the Tecton Proof System*

Deepak Kapur[1], David R. Musser[2] and Xumin Nie[1]

[1]Institute for Programming and Logics
Computer Science Department
State University of New York
Albany, New York 12222

[2]Computer Science Department
Rensselaer Polytechnic Institute
Troy, New York 12180

Abstract

The Tecton Proof System is an experimental tool for constructing proofs of first order logic formulas and of program specifications expressed using formulas in Hoare's axiomatic proof formalism. It is designed to make interactive proof construction easier than with previous proof tools, by maintaining multiple proof attempts internally in a structured form called a proof forest; displaying them in an easy to comprehend form, using a combination of tabular formats, graphical representations, and hypertext links; and automating substantial parts of proofs through rewriting, induction, case analysis, and generalization inference mechanisms, along with a linear arithmetic decision procedure. Further development of the system is planned as part of an overall framework aimed at supporting the kind of abstractions and specializations necessary for building libraries of generic software and hardware components.

*Partially supported by National Science Foundation Grants CCR–8906678, CDA–8805910, CCR–9308016, and CCR–9303394, and subcontract CB0204 of SRI Contract MDA904-92-C-5186 with The Maryland Procurement Office.

Contents

1 Introduction

Tecton (Greek for "builder") is a methodology and tool set for formal specification and verification of computational systems (both hardware designs and software) [19, 20, 23, 24, 21]. In formulating the goals of Tecton and designing its tools, we are seeking to combine many of the key advances in specification and proof technology, from both the authors' and many others' work on earlier specification language and proof systems. Beyond this, we also seek to *simplify* the use of formal methods, making them more accessible to non-experts and more easily applicable to non-trivial computational systems. A key to achieving these goals, we believe, is fostering more prominent uses of *abstraction.* By abstraction we refer not just to the popular notion of abstracting away from implementation details, as in the use of data abstractions in software development or structural abstractions in hardware design, but also to *behavioral abstraction.* By abstracting behavior and carefully engineering interfaces, we can produce software or hardware design *building blocks*, or *generic components* [32], that are much more easily composable and widely usable than the specialized components usually constructed in current practice. By applying formal methods to such generic components, we may attain further substantial benefits: (1) improvements in the cost-effectiveness of applying formal methods to software or hardware development, since the cost for a generic component, though high, can be amortized over its many uses; and (2) improved structure and documentation of the resulting software and hardware for purposes of coordinating development in large projects and simplifying maintenance and future enhancements.

Within this framework, the Tecton Proof System is an experimental tool for constructing proofs of first order logic formulas and of program specifications expressed using formulas in Hoare's axiomatic proof formalism [18]. In its present form it is noteworthy mainly for three features. One is the way it represents and manages proofs internally in flexible structures called *proof forests*, allowing records of multiple complete or incomplete proof attempts to be retained (an extension of the proof forest notion of [9]). The second is the structured external display format the system uses to present proofs to the user, using tables, graphics, and hypertext links. The third is the power of its mechanisms for automating many steps of proofs, which are based principally on earlier work on rewriting and mathematical induction methods from two previous systems, our own Rewrite Rule Laboratory (RRL) [27] and the Boyer-Moore prover [4].

In this paper we describe these proof structuring, displaying and automation features, including our motivations for some of the main design decisions. We must point out that currently the system does not provide full support for behavioral abstraction, although we have used it to carry out major parts of proofs about generic software components. The major weakness of the system is its specification language, which is essentially just traditional first order logic and Hoare formulas. However, we have designed a new language, also named Tecton [19], in which we attempt to unify and simplify previous research with similar goals, mainly drawing from research on abstract data types and from previous formal specification languages that emphasized behavioral abstraction, including mainly OBJ [13], Larch [16], and an earlier Tecton language design attempt [23, 24]. We have recently begun developing a Tecton Specification Processor that together with the Tecton Proof System should provide strong support for using formal methods on generic software and hardware design components.

The next section discusses representation of proof attempts in Tecton, including the central notion of a proof forest. In Section 3 we discuss the way the current Tecton system displays formulas and proofs, using (1) tabular formats to structure the display of first-order formulas, Hoare formulas, and programs, and (2) a combination of graphical layout of tree structures and hypertext links for proof tree display. Section 4 is an overview of the inference mechanisms supported, which are drawn mainly from RRL, including the use of inference rules for Hoare proof rules involving programs, inference mechanisms used to prove first-order formulas arising in specification analysis and in building theories, and the integration into the theorem prover of a decision procedure for universally quantified Presburger arithmetic with uninterpreted function symbols [26]. Section 5 contains brief examples from Tecton proofs, and Section 6 compares the main features of Tecton with other proof approaches. Finally, Section 7 returns to the discussion of planned extensions of the proof management system and the proposed development of the overall specification and verification environment.

2 Proof construction and management

The task of verifying the correctness or other semantic properties of computer programs or hardware designs often involves large, complex proofs, requiring the statement and proof of many lemmas and theorems. We believe that this task will become truly practical only when machine assistance can be effectively used even at the earliest stages of stating conjectures and attempting proofs.

2.1 The nature of proofs and proof construction

In books on logic, a formal proof is defined as a sequence of formulas with justifications for each formula. A "pure" proof involves two types of justifications: (i) a formula is an instance of an axiom schema, or (ii) a formula is obtained by a particular rule of inference applied on other preceding formulas in the sequence. In practice, such "pure" proofs from first principles tend to be very long, complex, tedious, and difficult to understand. Consequently, one rarely finds such proofs in the literature. Instead, proofs are hierarchically structured using derived rules of inference and meta-theorems, which, in addition to axiom schemas and rules of inference, are freely used as justifications. In order to present or understand a proof, its structure becomes crucial. Most complex proofs typically require human organization. Developing good proofs is an art much like writing good programs.

Computer generated proofs have an additional problem of opaqueness, because usually a computer proof is the result of an exhaustive search process (built into the heuristics or strategy used by a theorem prover intertwined with human guidance). For understanding such proofs, it is all the more essential that proof structure is highlighted by explicating the inference steps used in a proof attempt and whether a particular path of inference steps led to success or failure.

Ensuring that a computation indeed realizes a given specification is an iterative process. During the process of finding a proof, typically bugs are uncovered in a specification or in a program or hardware design implementing a computation. Most of the time, proof attempts exist in a partially completed state, and the user spends most of his or her effort trying to comprehend the current state of the attempt in order to guide the system towards a complete proof [28]. Multiple goals with incomplete proofs typically coexist, and it may sometimes be useful to maintain several distinct attempts to prove the same goal.

Much like software, successful proof attempts can be reused. Related theorems have related proofs. A proof of a related theorem can be obtained by slightly modifying inference steps used in a proof of another theorem. For example, we have encountered this phenomenon while doing proofs of searching, sorting and string matching algorithms. Proofs can be parameterized as well as generalized by identifying and abstracting common patterns of inference steps. This is especially evident while carrying out proofs about generic components.

Most verification systems do not provide adequate tools to deal with these issues. The structure of the proofs is buried in a style of linear representations most suitable for texts. The theorems and lemmas used in proofs are not readily available with the proofs and have to be looked up in an often large list of mostly irrelevant theorems and lemmas. Most systems still use a teletype or text-editor-buffer style interface. (See also [40] for a general discussion of improving interfaces to theorem provers.)

In Tecton our approach for escaping such limitations combines a structured internal representation of proofs and proof attempts with proof visualization methods using graphics, hypertext links, and tabular formats. For the remainder of this section we concentrate on the principles of our internal proof representation; the next section briefly describes the proof visualization issues.

2.2 Managing proofs and proof attempts

Formal proofs are often written as a linear sequence of formulas, starting with axioms and ending with the formula being derived, with annotations indicating which previous lines and which inference rules are being used. This representation

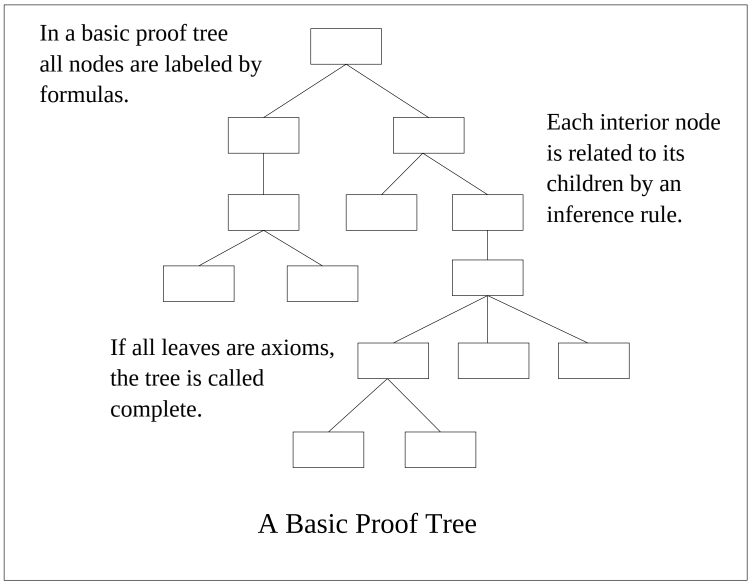


Figure 1: Structure of a Basic Proof Tree

is just a linearization of an underlying tree, with the formula at its root and axioms at the leaves, and with every non-leaf node related to its children according to an inference rule. Such a tree, illustrated in Figure 1, is an example of what we will call a *basic proof tree*. A postorder traversal of a basic proof tree produces the traditional linear sequence formal proof.

Tecton represents proof attempts as a forest of proof trees. The representation was chosen to be a tree structure because we believe it makes it easier to see the relation between the steps of the proof and because construction of a proof is often easiest if it proceeds from the goal (the root of the tree) to subgoals, repeatedly, until subgoals are reduced to trivialities. The representation used in Tecton is more elaborate than a basic proof tree in order to allow suppression of details and thus substantially reduce proof tree size, and also to allow for incomplete proof attempts, multiple proof attempts of the same formula, and the use of lemmas. The chosen representation, called a *general proof forest*—basically a set of "and/or trees" [2, pp. 39-42] such as are used in general problem solving strategies—is illustrated in Figure 2 through 4 and is defined precisely below. (An alternative of using a directed acyclic graph was also considered and rejected for reasons that are discussed later.)

2.2.1 Basic proof trees and general proof trees

Before defining general proof trees, we give the precise definition of a basic proof tree. Let $\mathcal{L}$ be a given logic consisting of a set of axioms and a set of inference rules.

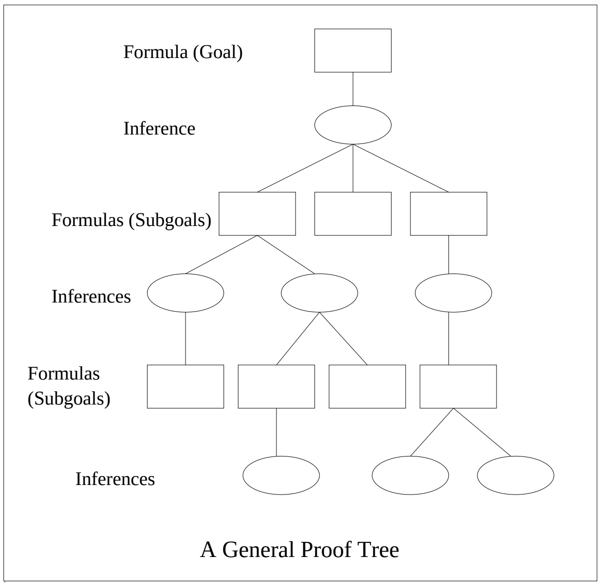


Figure 2: Structure of a General Proof Tree

Definition 1 *A* basic proof tree *(with respect to $\mathcal{L}$) is a finite tree in which each node is labeled with a formula, and for each non-leaf node there is an instance of an inference rule of $\mathcal{L}$ such that the children of the node are labeled by the premises of this instance and the node itself is labeled with the conclusion of the instance.*

Note that a basic proof tree does not necessarily contain a complete proof of the formula at its root; it does so only if all the formulas at its leaves are axioms of $\mathcal{L}$, and in that case it is called a *complete* basic proof tree.

Definition 2 *A* general proof tree *(with respect to $\mathcal{L}$) is a finite tree in which each node is either a formula node or an inference node, with the following properties:*

- *A formula node is labeled with a formula and has zero or more children, each of which is an inference node.*
- *An inference node is labeled with the name of an inference rule or a "basic inference mechanism" (defined below) and has one parent and zero or more children, each of which is a formula node.*
- *If the parent of an inference node is N and its children are $N_1, \ldots, N_k$, where $k \geq 0$, then there must exist a basic proof tree with the formula labeling N at its root and every leaf labeled either by an axiom of $\mathcal{L}$ or one of the formulas labeling $N_1, \ldots, N_k$.*

The last part of this definition is illustrated in Figure 3.

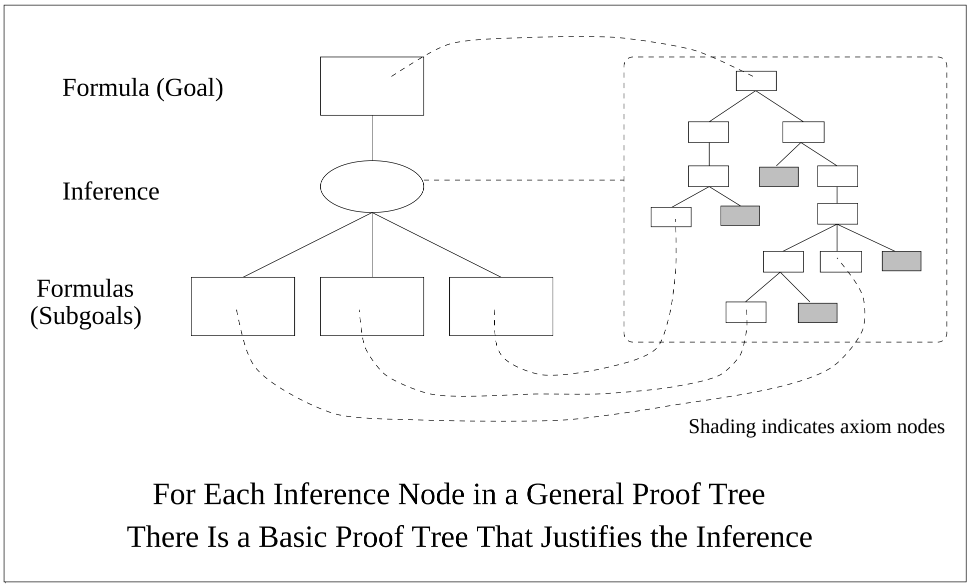


Figure 3: Correspondence between an inference node of a general proof tree and a basic proof tree

By this definition, the root of a general proof tree must be a formula node, and axioms do not explicitly appear in the tree. Thus an inference node with no children indicates there is a proof of its parent formula, given by a complete basic proof tree with the parent formula at its root. For an inference node with children $N_1, \ldots, N_k$, the existence of the basic proof tree means that there is a deduction of the parent formula from the formulas labeling $N_1, \ldots, N_k$.

This definition allows inference nodes to stand for arbitrarily large basic proof trees, but how much suppression of detail is actually done by this means is partly a matter of the availability of powerful inference mechanisms (discussed below) and partly a matter of how sophisticated the intended reader of the proof is assumed to be. (One might forego a powerful inference mechanism if the intended reader isn't known to be aware of that mechanism or, if aware, to be easily able to validate its use.)

2.2.2 Complete proofs in general proof trees

General proof trees contain steps of proofs, but not necessarily any complete proof. A simple case of a complete proof is that each formula node has a single child inference node, and the leaves of the tree are all inference nodes. More generally, we define inductively the case in which one or more complete proofs are present in the tree.

Definition 3 *Let F be a formula labeling a node N in a general proof tree $\mathcal{T}$. Then a subset of $\mathcal{T}$ is a* complete proof of F in $\mathcal{T}$ *if it consists of N, one child (an* *inference node) I of N, and a complete proof of every child of I. (The recursion terminates at inference nodes with no children.)*

Theorem 1 *Let $\mathcal{T}$ be a general proof tree, with respect to a given logic $\mathcal{L}$, and let $\mathcal{U} \subset \mathcal{T}$ be a complete proof of a given formula F in $\mathcal{T}$. Then F is provable in $\mathcal{L}$; i.e., there is a complete basic proof tree for F with respect to $\mathcal{L}$.*

Proof: By induction on the depth of $\mathcal{U}$. We omit the details, saving them for Theorem 2 below, which deals with an even more general notion of proof. □

A formula may have no proof (all proof attempts are incomplete), one proof, or more than one proof in a general proof tree, and complete proofs and incomplete proofs may be present in the same tree. The nodes of incomplete proofs could be pruned away once a complete proof is constructed, although there are situations in which one might want to retain them. For example, perhaps the only complete proof that has been found is very large, and one suspects that a way can be found to extend some incomplete proof attempt to a complete proof that will be smaller than the existing one.

Some pruning can be done automatically, as discussed in a later section.

2.2.3 General proof forests

As just defined, the notion of a proof of a formula in a general proof tree requires proofs to be self-contained. Instead, we would like to shorten proofs and make them more understandable by using other theorems as lemmas. To allow for this, we include in the notion of inference node the possibility of referencing other formulas as though they were axioms. Like axioms, these other formulas do not appear explicitly as labels of children of the inference node. Since they are not actually axioms, a proof of each of these formulas must be given elsewhere. We thus extend the notion of general proof tree as follows.

Definition 4 *A* general proof forest *(with respect a given logic $\mathcal{L}$) is a set of trees in which each tree is a general proof tree as previously defined, except for the following extension: the label of a leaf of the basic proof tree associated with an inference node may be, in addition to an axiom of $\mathcal{L}$ or a child node formula, a formula labeling any other node in the forest (possibly in the same tree), subject to the circularity restriction stated below.*

The formula and its node are said to be *used as a lemma* by the inference node.[1] (See Figure 4.)

[1] In using this terminology, which follows fairly common mathematical practice, we must keep in mind that "lemmas" are really conjectures and might be invalid. When we need to make the distinction, we speak of "unproved lemmas" (conjectures without complete proofs) or "proved lemmas" (conjectures that have been proved and are therefore valid) [9].

Definition 5 *Define on a proof forest $\mathcal{F}$ a directed graph $\mathcal{G}$, called the* induced dependency graph *of $\mathcal{F}$, by including a directed arc from each node of $\mathcal{F}$ to each of its children and, additionally, from each inference node to any node used as a lemma.*

Circularity Restriction The use of lemmas in a general proof forest must be such that the induced dependency graph is acyclic (thus, a dag).

Informally, we see that this restriction rules out circular proofs; in the proof of the meta-theorem below justifying the notion of general proof forest, the restriction is essential to the induction argument.

As with proof trees, a proof forest might or might not contain any complete proofs.

Definition 6 *Let F be a formula labeling a node N of a proof forest $\mathcal{F}$. Then* a complete proof of F in the forest $\mathcal{F}$ *is a subgraph $\mathcal{H}$ of the induced dependency proof graph $\mathcal{G}$ of $\mathcal{F}$ such that:*

1. *$\mathcal{H}$ is rooted at N,*
2. *each formula node in $\mathcal{H}$ has exactly one arc of $\mathcal{H}$ emanating from it,*
3. *for every inference node I included in $\mathcal{H}$, all of the arcs emanating from it in $\mathcal{G}$ and all of its children are included in $\mathcal{H}$.*

Informally speaking, the first condition says that the proof represented by $\mathcal{H}$ is a proof of the desired formula, the second condition implies that there are no subgoal formulas with incomplete proofs, and the third condition ensures that all dependencies are accounted for. Since $\mathcal{H}$ is a subgraph of a dag, it must itself be a dag, and thus there can be no circularities in the proof. The following theorem and proof show that the notion "complete proof of a formula in a proof forest" does indeed correspond to more familiar notions of formal proof.

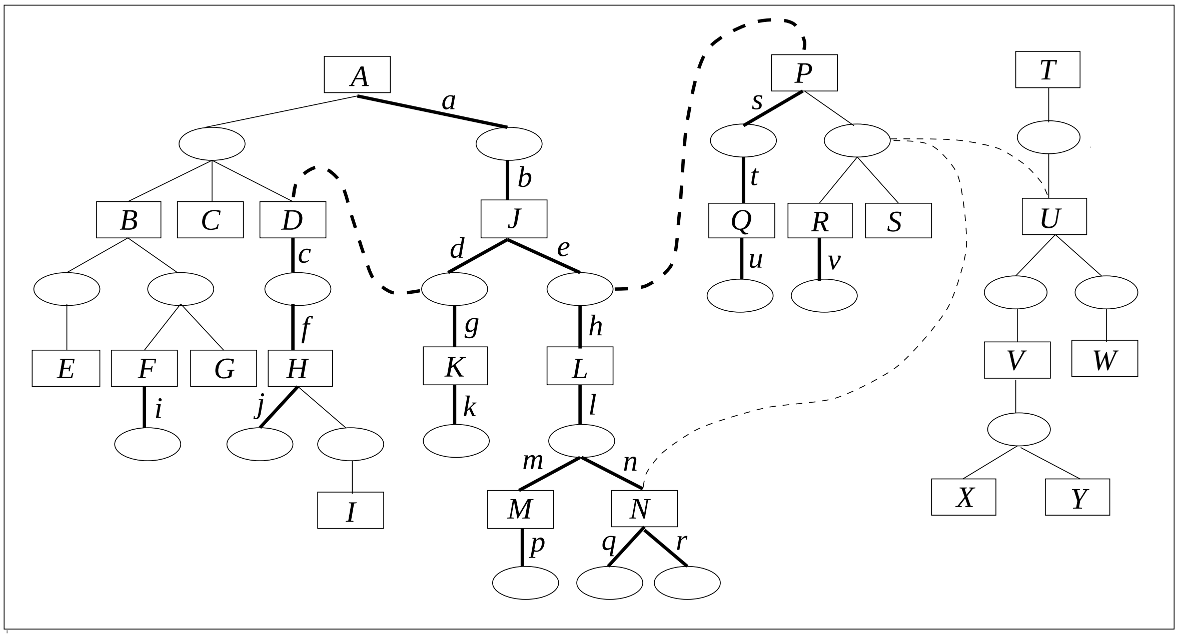


Figure 4: A general proof forest, with examples of complete proofs. Dashed lines represent arcs included in the induced dependency graph corresponding to lemma uses. Heavy lines (solid or dashed) represent arcs that belong to complete proofs in the forest. For example, formula D has a proof containing arcs c, f, and j, and formula A has three distinct complete proofs in the forest. One proof contains arcs a, b, d, g, and k, plus the lemma-use arc to formula D and its proof. The second proof of A contains arcs a, b, e, h, l, m, n, p, and q, and the lemma-use arc to formula P and its proof with arcs s, t and u. The third proof of A is the same as the second except that it contains arc r instead of q.

Theorem 2 *Let $\mathcal{F}$ be a general proof forest, with respect to a given logic $\mathcal{L}$, and let F be a formula labeling a node N in $\mathcal{F}$. Suppose $\mathcal{H}$ is a complete proof of F in $\mathcal{F}$. Then F is provable in $\mathcal{L}$; i.e., there exists a complete basic proof tree for F with respect to $\mathcal{L}$.*

Proof: By induction on the maximum path length from N in the graph $\mathcal{H}$. Since $\mathcal{H}$ is a complete proof of F in $\mathcal{F}$, N must have a child node I to which it is connected by the unique arc of $\mathcal{H}$ emanating from N. Let $N_1, N_2, \ldots, N_k$ be the children of I in $\mathcal{F}$; again these nodes and arcs from I to them must be in $\mathcal{H}$. There must also be arcs from I to any nodes $N_{k+1}, N_{k+2}, \ldots, N_n$ of the forest used as lemmas. By the definition of general proof tree, there must be a basic proof tree T_0 with N at its root and such that every leaf node is either labeled by an axiom or is one of $N_1, N_2, \ldots, N_n$. If $n = 0$ (no subgoal formulas or lemmas), then all of the leaves of T_0 are axioms and T_0 is a complete basic proof tree with N at its root, and we are done.

Otherwise, for each node N_i, the maximum length of a path in $\mathcal{H}$ starting at N_i is (two) less than the maximum path length from N, so by the induction hypothesis there exists a complete basic proof tree T_i for $N_i, i = 1, \ldots, n$. By replacing each N_i node in T_0 with its complete proof tree T_i, we obtain a complete basic proof tree for N. $\Box$

A reasonable alternative to the notion of general proof forest would be a formulation of proof structure directly as a dag, such as the induced proof graph $\mathcal{G}$ described above. We prefer the proof forest notion because we believe it corresponds more closely to the traditional concept of proof in textbooks and journals, in that when a lemma is used it is most often cited by name rather than by explicitly writing out the statement or proof of the lemma in-line. In a well-structured textbook or journal proof the use of a lemma at a particular point in a proof usually indicates that the author considers it best not to divert one's attention away from the current proof to the proof of the lemma; rather, the lemma and its proof should be examined separately and independently. We have thus relegated the role of the induced dependency graph to that of a meta-level tool that aids in justifying the correctness of our formulation of general proof forests.

3 Proof visualization

The general proof forest notion described in the previous section is a fundamental concept that we expect to retain intact as further development of the Tecton Proof System proceeds. By contrast, we regard many aspects of the visual representation used in the current Tecton Proof System as tentative design decisions for purposes of experimentation; they may be revised, perhaps extensively, in future versions of the system.[2] We therefore describe them in this paper only in enough detail to make the examples in Section 5 intelligible.

3.1 Graphical layout of proof trees and forests

The current Tecton system displays the graphical layout of a complete or partial proof tree on a series of pages, with hypertext links included within the pages to pages containing continuations of the same tree or other trees in the proof forest.

Because of some special characteristics of proof trees, the format we currently use in Tecton to display proof trees is a little different from that indicated in Figure 2. Since it is common for proof goals to have only one child, we save space by placing the inference node directly below its parent and omitting the line connecting it to its parent. We also omit the enclosing oval, just using the text item that names the inference rule or mechanism. If there is more than one child of a goal we duplicate the goal node on another page and show an alternative inference in the same position as the one on the first page, directly below the goal. A hypertext link to the alternate is placed on the first page next to the inference text item.

The layout of the subgoals of an inference is also a little unusual (see Figure 5). This format is used for two reasons. First, since goals are often expressed with long text items, a format that easily accommodates such items is desirable. Second, a frequently used inference mechanism is *reduce,* which produces only one subgoal. In this case ($n = 1$ in Figure 5), the subgoal is aligned directly below the parent goal; no horizontal space is lost to indentation, as might be the case with more conventional tree layouts.

If there is not enough space on the current page to display all the subgoal formula nodes below an inference node, a *continuation marker* is created in place of the inference node, textually indicating on which page the subgoals will be displayed and linked to that page with a hypertext link. Continuation markers are created on demand as the proof proceeds. On the new page, an item is created that links with and textually indicates the page from which the new page is continued.

A user can use the sibling and continuation links to browse through the proofs quickly and to examine different parts of the proofs.

3.2 Tabular formats and symbols

The examples in Section 5 show the way the current Tecton system displays formulas in a tabular format, designed to reduce the number of logical connectives necessary in comparison with traditional notations of formal logic. Details and

[2]We are also developing alternative visual representations as part of a project to provide a graphical/hypertext interface to SRI's PVS proof-checker [35].

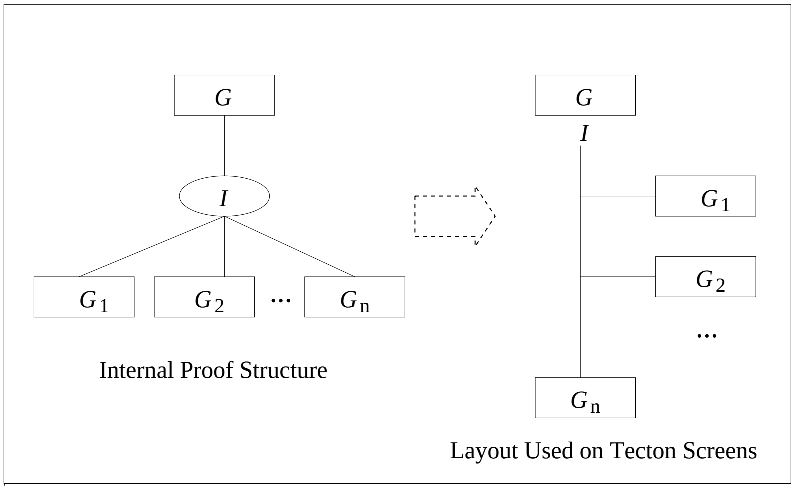


Figure 5: Layout of Inference Node and Subgoals

motivation for the format are sketched only briefly here; further detail may be found in [21, 22].

3.2.1 First order logic formulas

Formulas of first order logic are displayed in two column tables, with each row of a formula table representing an implication $\mathcal{A} \supset \mathcal{C}$, where $\mathcal{A}$ (assumptions) and $\mathcal{C}$ (conclusions) are the conjunctions of the atomic formulas in the left and right columns of the row, respectively. For example, the basic "laws of equality" can be expressed as

	$x = x$
$x = y$	$y = x$
$x = y$ $y = z$	$x = z$

The second row represents $x = y \supset y = x$. The third row represents $x = y \wedge y = z \supset x = z$. If an entry is blank, it is the same as if it contained the boolean value *true*. Thus the first entry represents $x = x$ (since this is same as $true \supset x = x$).

3.2.2 Hoare formulas

A Hoare formula is displayed in a three column row of a formula table. The row represents $\{\mathcal{P}\}\ \mathcal{S}\ \{\mathcal{Q}\}$, where $\mathcal{P}$ and $\mathcal{Q}$ are the formulas represented by the left and right columns of the row, respectively, and $\mathcal{S}$ is a programming language statement represented in the middle column. $\mathcal{P}$ and $\mathcal{Q}$ represent, respectively, a *precondition* and *postcondition* for the statement. The statement $\mathcal{S}$ is represented by a string $\mathcal{S}'$ which may contain symbols that refer to program statements or expressions that are displayed in another table, called the (associated) Program Table. By substituting these statements or expressions for the corresponding symbols in $\mathcal{S}'$, one obtains the represented statement $\mathcal{S}$.

Preconditions and postconditions are also called (program) *assertions.* The meaning of a row of the table is the usual conditional correctness semantics of Hoare formulas: if the precondition is true of the program state σ before the statement is executed, and execution of the statement terminates producing a new program state σ', then the postcondition is true of σ'. For example, the Hoare formula

$x > y$	$x := x + 1$	$x > y + 1$
$z = a$ $z > 0$ $x = 1$	while $z \neq 0$ do $x := x * y;\ z := z - 1;$ endwhile	$x = y^a$

is the conjunction of two Hoare formulas, the first about an assignment statement $x := x + 1$ and the second about a while statement that computes an integral power y^a.

4 Inference rules and mechanisms

A proof system for first order logic is typically defined to have only a few primitive inference rules, such as *modus ponens*

$$A, A \supset B \vdash B.$$

In principle, proofs could be done using only these rules. But most proofs would have to be uncomfortably large if constructed out of such simple building blocks. In order to reduce the size of proofs, it is convenient to allow the use of *derived rules of inference*, which are rules of the form

$$A_1, \ldots, A_n \vdash B \quad (n \geq 1)$$

where there exists a basic proof tree (see Section 2.2.1) with B at its root and $A_1, \ldots, A_n$ at the leaves. The use of such a rule thus merely abbreviates the corresponding proof tree to a tree with nodes only for B, leaves $A_1, \ldots, A_n$, and the inference node (referencing the derived rule) in between. A derived inference rule is analogous to a macro used in programming to generate a larger piece of code.

Tecton goes further in proof abbreviation than just using derived rules of inference. In a general proof tree an inference node may refer either to a primitive inference rule, a derived inference rule, or a basic inference mechanism, as described in the following subsections.

4.1 Primitive and derived inference rules for Hoare formulas

In the current Tecton Proof System, the only explicit use of primitive and derived rules on inference nodes are applications of Hoare rules [18] for eliminating program structure from Hoare formulas. Hoare formulas are used to express properties of programs, by relating program statements to assertions expressed in first order logic. Hoare inference rules are systematically used to reduce a Hoare formula goal into subgoals with simpler program constructs, until subgoals are eventually produced that are first order logic formulas. The Hoare rules used in Tecton are mostly conventional and their soundness is easily justified in terms of a formalization of the usual conditional correctness semantics. The current Tecton Proof System has inference rules for the following statement types [22]: *assignment*, *composition*, *conditional*, *while*, *declare*, *release*, *abort*, *null*, *choose* (nondeterministic choice), and *exchange*. Some of these are illustrated in the sample proof in Section 5. The main departure from Hoare's formulation is in the use of derived inference rules for assignment statements and conditional statements that allow programs to be written with fewer intermediate assertions.

4.2 Inference mechanisms

Definition 7 *A* basic inference mechanism *is an algorithm that takes a formula as input and produces as output a finite sequence of formulas, such that there exists a basic proof tree whose root is labeled with the input formula and whose leaves are labeled with the output formulas or with axioms or lemmas.*

Corresponding to any particular use of a basic inference mechanism is an anonymous derived rule of inference; one might say that the mechanism generates derived rules "on the fly." This is just an observation; the definitions and theorems previously given about general proof forests have already justified the claim that complete proofs within a proof forest correspond to more conventional notions of proof.

Definition 8 *A* general inference mechanism *is a heuristic procedure that takes as input a designated formula node in a general proof forest and extends the forest by addition of new general proof trees rooted at the node and, possibly, new separate general proof trees. The extensions must preserve general proof forest properties.*

This definition, together with Definitions 2, 4, and 7, implies that each inference node I constructed by a general inference mechanism represents the use of either

- an instance of some inference rule (primitive or derived) whose conclusion is the formula labeling the parent of I and whose premises are the formulas labeling the children of I, or

- some basic inference mechanism M in which the formula labeling the parent of I is the input to M and the formulas labeling the children of I are the outputs of M.

The definition allows a general inference mechanism only to extend a proof forest; it may not delete or replace any trees or subtrees. (A separate notion of pruning is discussed below.) The definition also implies that no cycles may be introduced in the proof forest's induced dependency graph.

The case of extension with separate trees occurs when a general inference mechanism automatically generates new lemmas and attempts to prove them. Unlike basic inference mechanisms, a general inference mechanism might not always terminate (thus we call them heuristic procedures rather than algorithms). In Tecton's interactive environment the user is provided with a way of manually interrupting execution of a general inference mechanism.

The nomenclature of "basic" versus "general" inference mechanisms has been chosen in analogy to the distinction we make between basic versus general proof trees. Basic proof trees do not appear explicitly in general proof trees; an inference node in a general proof tree just represents the existence of a basic proof tree. Similarly, basic inference mechanisms do not affect a proof tree directly, but they are used by general inference mechanisms, which extend an existing proof forest with new subtrees or separate trees.

The following subsections describe the main inference mechanisms, both basic and general, currently used in Tecton.

4.2.1 General inference mechanism for Hoare formulas

For a Hoare formula goal, Tecton invokes its Hoare Rule Mechanism, a general inference mechanism that begins by applying a single Hoare inference rule as determined by the program construct of the formula, producing an inference node labeled with the name of that construct and children labeled with Hoare formulas or first order logic formulas. If no child is a Hoare formula node, the Hoare Rule Mechanism terminates; otherwise it applies itself recursively to each Hoare formula node.

All of the other Tecton inference mechanisms are applicable only to first order logic formulas, not to Hoare formulas.

4.2.2 Reduction

One of the most important basic inference mechanisms used in Tecton is *reduce*, an algorithm that makes a sequence of rewrites using rewrite rules, producing as its output the final formula of the rewrite sequence. The rewrite rules used are unconditional or conditional rules corresponding to proper axioms or lemmas. In a first order logic with equality, rewriting is justifiable by a basic proof tree, since the laws of equality (reflexivity, symmetry, transitivity, substitution) are axioms or theorems of the logic.

Rewrite rules are formed by Tecton using a term ordering to determine which side of an equation to take as the left hand side of a rule so that the rewriting sequences generated by such rules are always finite. The ordering used is called *lexicographic recursive path ordering* (lrpo) [8]. (For equations not orientable by lrpo, it is possible for a user to manually orient equations into rewrite rules; but, then there is a danger of rewriting not terminating. Tecton generates a warning if a large number of rewrites are performed in *reduce*.) Additional rewrite rules may be generated by Tecton using a variant of the Knuth-Bendix procedure (currently Tecton includes implementations of several variants).

Also built into *reduce* are transformations and simplifications based on the logical axioms used by Tecton. Use of these transformations is *not* indicated on the inference node. For conditional rewrite rules, Tecton uses *contextual* rewriting; examples illustrating contextual rewriting and its power are discussed in detail in [42].

In case the resulting formula produced by *reduce* is *true*, there is no subgoal of the reduce inference node, and it is labeled with "finish by ..." instead of "reduce by ...."

Note that reduction includes the important case of applying another theorem as a lemma. In the current system, the inference node cites the theorem by a number assigned to the theorem in a table called the Rules table. Rules tables list all rules used in different applications of the reduction inference mechanism.

The reduction mechanism is illustrated in the proofs in Section 5.

4.2.3 Case analysis and splitting

Tecton supports two types of conditional rules. The first type is a rule of the form $l \rightarrow r \quad \mathit{if} \;\; c_1 \;\wedge\; \cdots \;\wedge\; c_k$, which means that the rule is applicable if an instance of l using a substitution σ can be identified such that each $\sigma(c_i)$ reduces to true (possibly by rewriting). The contextual rewriting mechanism is used for this. The second type of a conditional rule is of the form: $l \rightarrow \mathit{cond}(b, r_1, r_2)$, where r_1 and r_2 could also have occurrences of *cond*. Such a rule is treated as a unconditional rule and is applied whenever an instance of l is identified. Later, if a goal cannot be reduced any further, and *cond* appears in a goal, then the *case* inference mechanism applies, producing subgoals based on the first argument of *cond*, which is a boolean value. There are two subgoals, one in which the first argument is replaced by *true* and the other in which the first argument is replaced by *false*. This kind of restricted case analysis has been found quite useful.

Another basic inference mechanism is *split*, which eliminates boolean operators, $\wedge$ and $\vee$, in a goal by generating necessary subgoals. For example, given a goal $\mathcal{A} \;\supset\; C_1 \wedge C_2 \wedge \cdots \wedge C_n$, the split mechanism produces subgoals $\mathcal{A} \;\supset\; C_i$, for $i = 1, \ldots, n$.

4.2.4 Induction

Tecton uses the *cover set induction* mechanism [41]. Many heuristics are used to choose an induction variable. In fact, the choice of an induction variable is determined by selecting a subterm whose definition is used to generate an induction scheme for the data structure(s) appearing in the domain of the subterm. If the first choice of an induction variable and induction scheme do not succeed, then Tecton backtracks and attempts another choice until no more choices can be made. The induction mechanism is thus an example of a general inference mechanism as it may generate a general proof tree with multiple proof attempts.

Consider a subterm occurring in a goal G, and consider the definition, by rewrite rules, of the outermost function symbol of the subterm. Left sides of the rules in the definition constitute a cover set.

Suppose that terms $t_1, \ldots, t_k$ constitute a cover set for performing induction on subterm $f(x_1, \ldots, x_n)$, where each t_i has f as its outermost symbol. In general, $k + 2$ subgoals should be[3] generated using a well-founded ordering $\prec$:

1. Induction subgoals: Corresponding to each t_i a subgoal G_i, generated from the goal G where $f(x_1, \ldots, x_n)$ is replaced by t_i, with an induction hypothesis as an additional assumption for the subgoal. That is, G_i is obtained from $G(f(x_1, \ldots, x_n))$, producing $G(t'_i) \supset G(t_i)$, where $t'_i \prec t_i$. The case when $G(t'_i)$ is true is the basis step. In general, there can be as many induction hypotheses as needed insofar as the subterms used in place of t_i are smaller than t_i with respect to $\prec$ (also see item 3 below).

2. Completeness: A subgoal for proving that this is a complete cover set for f. That is, the union of the sets of ground instances of $t_1, \ldots, t_k$ is equivalent to the set of ground instances of $f(x_1, \ldots, x_n)$, when the x_i's are substituted by constructor ground terms of the type of x_i's.

3. Well-foundedness: A subgoal $t'_1 \prec t_1 \wedge \ldots \wedge t'_k \prec t_k$. Often, t'_i is a subterm from the right-hand side of the rule defining f whose left side is t_i. Since these rules are oriented using lexicographic recursive path ordering (lrpo) we can finish this subgoal by saying that the proof follows from the well-foundedness of lrpo. However if a rewrite rule in the definition was manually oriented and subterms t'_i from its right side with f as the outermost symbol are used for generating induction hypotheses, then it is essential to ensure for each subterm that that $t'_i \prec t_i$.

The cover set induction mechanism is illustrated in the first proof in Section 5.

[3] Currently Tecton only generates the induction subgoals, not the completeness and well-foundedness subgoals.

4.2.5 Generalization

If a goal cannot be proved by the reduction, case and split mechanisms, then it is often the case that it must be proved by induction. Before attempting a proof by the cover set induction mechanism, the goal is analyzed to examine whether a more general, and perhaps more useful, conjecture can be attempted.

The *generalize* mechanism in Tecton currently performs two types of generalization. The first attempts to drop an assumption from a conditional goal. If a literal in a conditional goal is expressed using variables that do not occur in the rest of the goal, it is dropped. The second type of generalization mechanism looks for common subterms appearing on the two sides of an equational goal or in the assumptions and conclusions of a goal. It produces a subgoal in which all such subterms are replaced by distinct variables. It should be noted that the generalize mechanism can generate a subgoal formula that is invalid, even if the goal was valid. When this happens it is necessary to try an alternative generalization or another kind of inference step.

In the second type of generalization, Tecton first generates a subgoal with as many generalizations of subterms to variables made as possible. If it does not succeed in proving that formula, other generalizations in which only some of the subterms are generalized to be variables are attempted. This is repeated until either a generalization can be proved or every possible generalization fails. In the latter case, the original goal is then tried. In this way, Tecton automatically makes multiple proof attempts, backtracking when it does not succeed.

If a subterm (or subterms) being generalized appears in the left side of an unconditional rule that is some theorem already proved or a part of a definition, then the generalization mechanism uses this unconditional rule to generate a *constraint* on the new variable being introduced. This constraint is generated only if no variable in the subterm being generalized appears elsewhere in the rule. For instance if t is a subterm being generalized to a variable u and there is an unconditional rule $l \rightarrow r$ in which t appears in l at position p, and no variable in t appears elsewhere in the rule, then Tecton generates a constraint $l' = r$, where l' is obtained from l by replacing t with u at position p.

The generalization mechanism is illustrated in the first proof in Section 5.

4.3 Pruning and recycling

An inference rule or mechanism is called *validity-preserving* if from a valid formula it can only produce valid formulas as children. The inference mechanisms and rules used by Tecton are validity-preserving except (1) the generalize mechanism, as already noted, and (2) instances of reduction in which an unproved lemma is used. The distinction is important to another aspect of proof management: *pruning of failed proof attempts*.

Suppose a formula node F is marked as invalid, as might be detected by finding a counterexample (for example, in the Tecton induction mechanism). If F has a parent inference node I and grandparent formula node G, and I is validity-preserving, then G can also be marked as invalid, and invalidity can thus be propagated toward the root, stopping when encountering a non-validity-preserving inference. Then any maximal subtree $\mathcal{T}$ rooted at a formula node marked as invalid can simply be removed (some portions may be saved, as discussed below). For example, in Figure 4, if all of the inferences in the leftmost tree are validity-preserving except the root node of its left subtree, and G is invalid, then so is B, and the entire tree rooted at B can be removed.

In fact, even more pruning can be done: if $\mathcal{T}$ has parent I the entire subtree rooted at I can be removed. (I represents a non-validity preserving inference that did in fact lead to an invalid subgoal. The parent of I, though, might be valid and might have other inferences descending from it.) Continuing the above example from Figure 4, the entire left subtree of the leftmost tree could be removed (though some parts can be salvaged, as discussed below).

When a formula node is marked as invalid, any uses of that node as a lemma must also be invalidated. Any inference in which such a use occurs should be marked as being dependent on an invalid lemma, but there is no propagation beyond the inference. While it would be proper to delete the subtree rooted at the offending inference, in many cases such a proof can be salvaged by restating the lemma, perhaps by adding an assumption that can in fact be discharged. We thus plan to provide a means to change the lemma use reference to another formula node in the forest, one that is already existing or newly created.

On the other hand, suppose a proof of a formula is completed and that formula has been used as a lemma in other proof attempts (see footnote 1). By updating the status of inference nodes that use it, an inference may change status from non-validity-preserving to validity-preserving (its status might be unchanged if it depends on other unproved lemmas). If that happens, a further check can be made of whether the inference has an invalid child formula; if so, invalidity can be propogated to the inference node's parent and perhaps beyond.

A subtree removed by pruning might actually contain complete proofs of some of the formulas at its interior nodes. In the context of the general proof forest notion, it makes sense to extract a removed subtree's maximal subtrees that contain complete proofs and establish them as separate trees in the forest. Again considering Figure 4, if the left subtree of the leftmost tree is removed, the subtrees rooted at nodes labeled D and F can be retained. Other subtrees that do not have complete proofs but are referenced as lemmas should also be saved. For example, in Figure 4, if the rightmost tree is removed, the subtree rooted at the node labeled U should be retained since U is used as a lemma.

This *recycling* of proofs is easily implemented by a preorder traversal looking for formula nodes that are marked as having a complete proof or as being used as a lemma. Note that references that some inference nodes may make to other formula nodes used as lemmas are unaffected by this recycling, assuming that the identities of formula nodes are unchanged by pruning or recycling operations (as is the case, for example, if a linked representation of trees is used and a subtree is removed by unlinking, rather than by copying all but the subtree).

4.4 Linear arithmetic decision procedure

Because of the primary importance of numbers in the mathematics of computer programming, we have built into the Tecton reduction mechanism a decision procedure for a subclass of *Presburger arithmetic* with uninterpreted function symbols. Pure Presburger formulas are built using natural numbers, variables over natural numbers, addition, multiplication by constants, the usual arithmetical relations ($<, \leq, >, \geq, =$), and the first-order logical connectives (in fact, we consider formulas over integers, and natural numbers are treated as integers ≥ 0). The built-in procedure eliminates the need to explicitly state definitions of addition, subtraction, $\leq, <, >, \geq$ as well as transitivity axioms for $\leq$. In Tecton, formulas may include other function symbols whose properties are being given using rewrite rules outside the Presburger theory. The subclass decided consists of Presburger formulas that, when placed in prenex form, contain only universal quantifiers. Boyer and Moore incorporated a similar rational-based procedure into their heuristic theorem prover [5]. Our implementation is heavily influenced by the work of Boyer and Moore.

The core of the decision procedure is Fourier's algorithm for deciding linear inequalities over the rationals. The extension attempts to generate implicit equalities that are transformed into rewrite rules for simplification and elimination. This decision procedure and contextual rewriting mechanism in Tecton use each other as mutually recursive procedures. The linear procedure may use the reduction mechanism to derive additional relations among integer terms; the reduction mechanism may call the linear procedure to establish the conditions of a conditional rewrite rule before the rule can be applied. This close interaction makes our procedure different from and more useful than stand-alone procedures such as discussed in [39]. This interaction by equality sharing is a powerful method to combine different decision procedures [33] and has made the linear procedure more useful.

Tecton does not implement a complete decision procedure for linear arithmetic. No attempt is made to check whether any derived equality on integer terms is satisfiable. Also, linear inequalities may be satisfiable over the rationals, but they may not be satisfiable over the integers; no attempt is made to determine whether a satisfying integer assignment to variables can be generated from the intermediate inequality constraints generated by Fourier's algorithm. For completeness, it is also essential to superpose subterms in a goal with the defining rewrite rules of interpreted function symbols, but that is also not implemented. A detailed description of the decision procedure and related issues is given in [26].

4.5 Overall inference mechanism

We can now describe the overall general inference mechanism Tecton uses to attempt to prove a formula. If the formula is a Hoare formula, Tecton first uses its Hoare Rule Mechanism to reduce a Hoare formula goal into subgoals with smaller program constructs until all subgoals are eventually reduced to first-order logic formulas.

Each such first-order formula (or any other first-order formula manually entered as a goal) is subjected to the split mechanism, and then the reduction mechanism, during which rewriting and the linear arithmetic procedure are invoked in a mutually recursive style. Case analysis is then performed, if applicable.

When it becomes impossible to apply any of the split, reduction, or case analysis mechanisms, the generalization mechanism is used. After that the cover set induction mechanism is applied to generate subgoals corresponding to the basis case(s) and the induction steps. Each of these subgoals is treated as a new goal, and the above steps are repeated. In the generalization as well as the cover set induction inference mechanisms, multiple proofs may be attempted when failure is detected or it is realized that pursuing a particular path is not likely to succeed. In that case, the inference mechanism automatically backtracks and tries other possible choices.

5 Examples of Tecton proofs

We give two examples that together illustrate some of the main features of the Tecton proof system. The first example is a proof, found automatically by Tecton, of a lemma about the integer remainder function. The proof found by Tecton is shown in Figures 6 and 7, which are pages produced by a prototype user interface implemented using KMS (Knowledge Management System) [1], a commercial hypertext system. The proof contains instances of the following inference mechanisms: cover set induction, reduction, generalization, and splitting. The linear arithmetic procedure was also used as part of the reduction mechanism. The axioms defining remainder and other integer functions are shown in the Rules tables on pages in which they are used. The Parts tables show definitions of symbols introduced automatically by the display algorithm to keep the formula nodes from becoming too large.

The second example shows part of a Tecton generated proof of conditional correctness of an efficient quotient-remainder procedure. Preceding this proof the system had been directed to prove about fifteen lemmas about integer operations (including the remainder lemma of the first example). The right half of Figure 8 is a Program Table that was constructed by the system from an ASCII text version of the program prepared by the user. The user input included pre- and post-conditions and invariants for loops. The left half of Figure 8 and Figures 9 through 11 show the application of the the Hoare Rule Mechanism to reduce the original Hoare formula goal to a set of ordinary first-order logic formula goals, and the use of splitting and reduction to finish these goals.

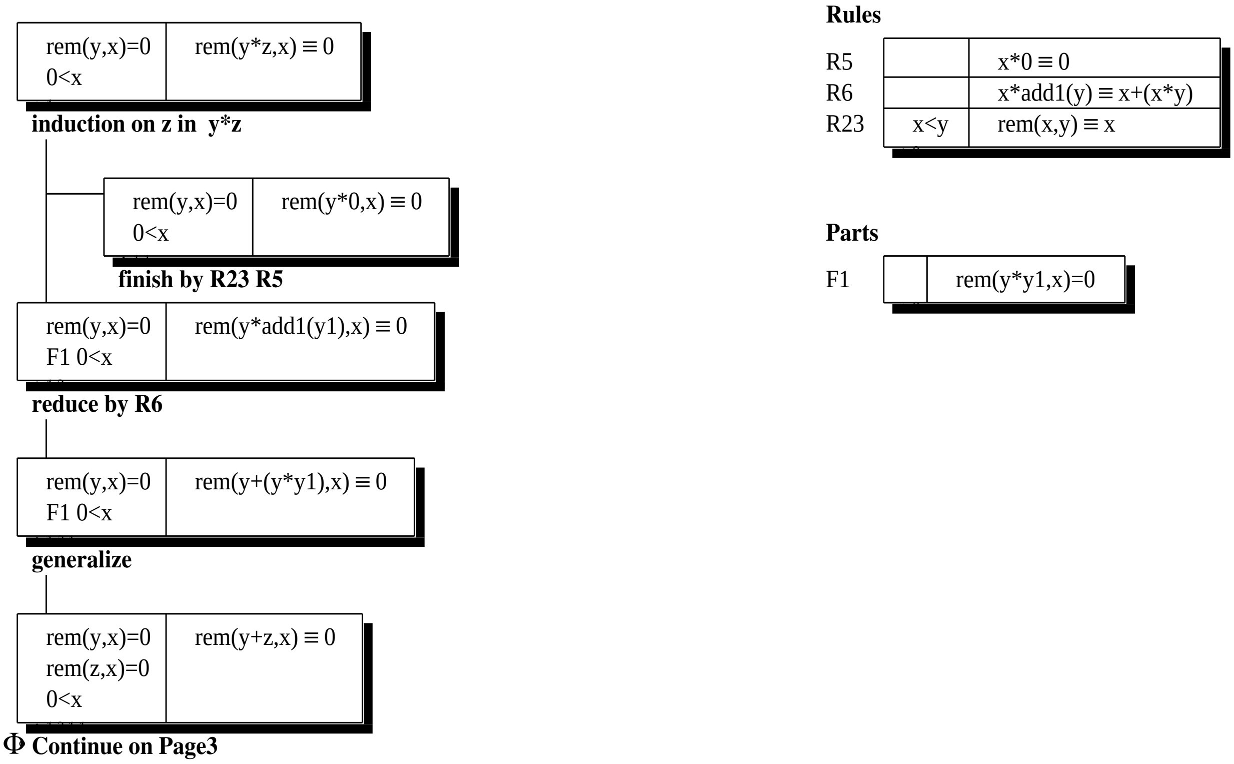
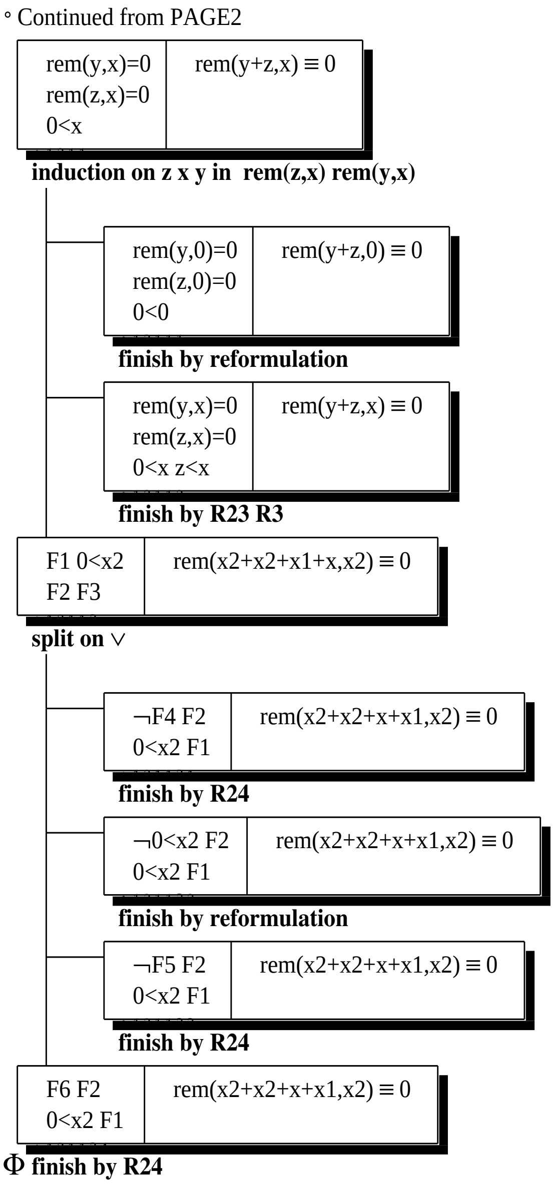


Figure 6: Beginning of the proof of a lemma about the remainder function, rem.



Rules

R3		x+0 ≡ x
R23	x<y	rem(x,y) ≡ x
R24		rem(y+x,y) ≡ rem(x,y)

Parts

F1		rem(x2+x1,x2)=0
F2		rem(x2+x,x2)=0
F3	F4 0<x2 F5	rem(x1+x,x2) ≡ 0
F4		rem(x1,x2)=0
F5		rem(x,x2)=0
F6		rem(x+x1,x2)=0

Figure 7: Completion of the proof of a lemma about the remainder function, rem.

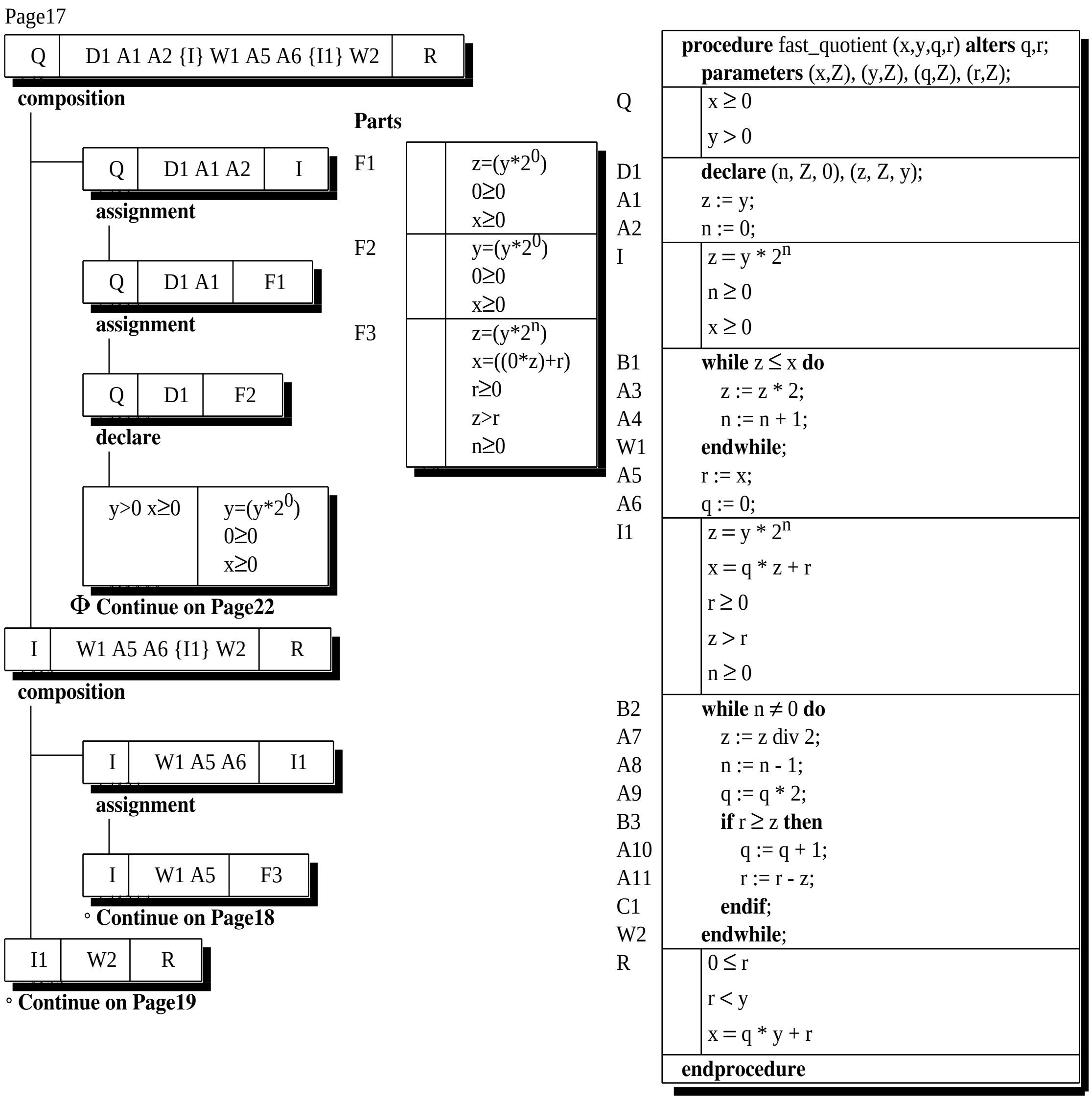


Figure 8: Program Table and beginning proof steps for an efficient division algorithm

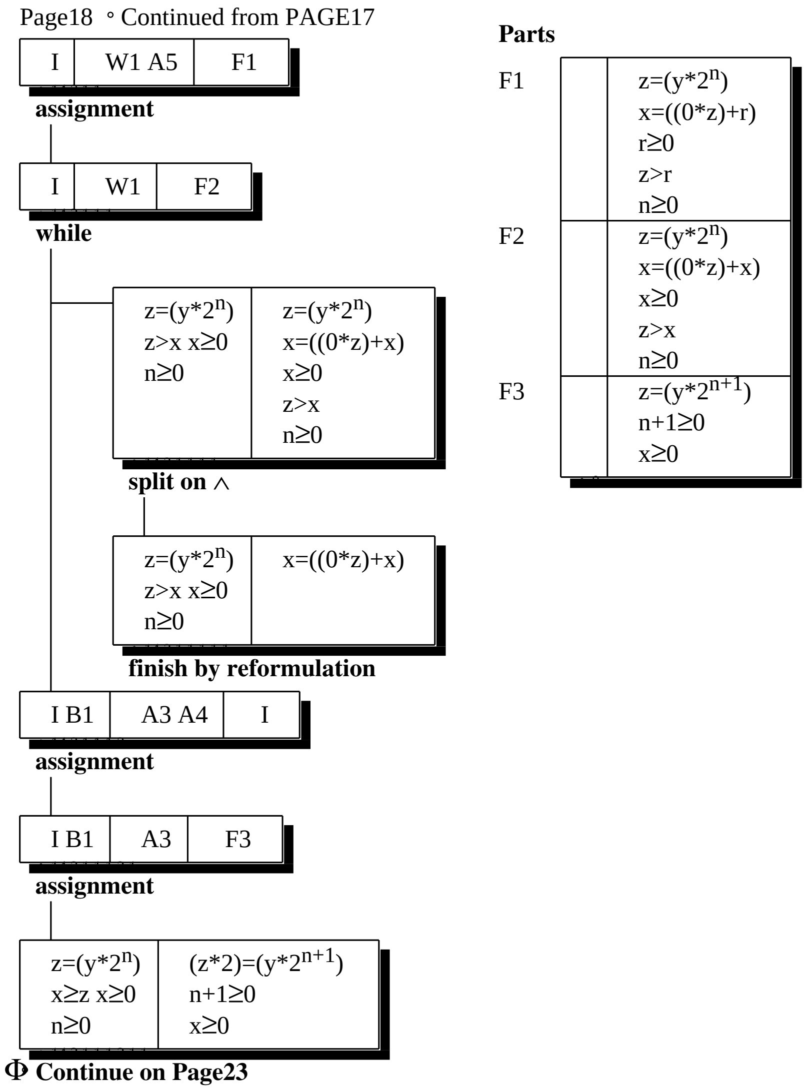


Figure 9: Continuation of the proof for the quotient-remainder procedure, showing the use of Hoare rules to eliminate the first while statement and the use of an inference mechanism, "by reformulation" (part of the reduction mechanism and linear arithmetic).

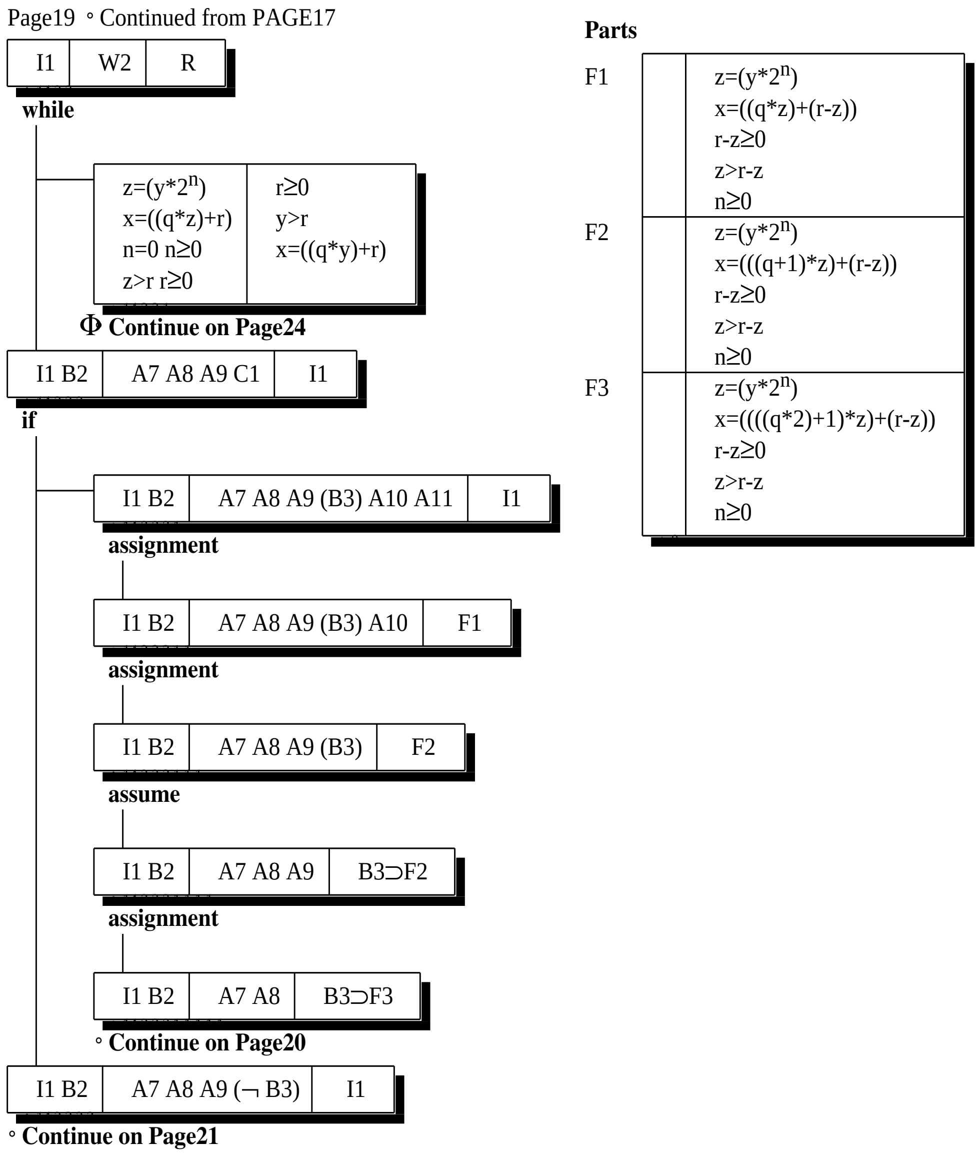
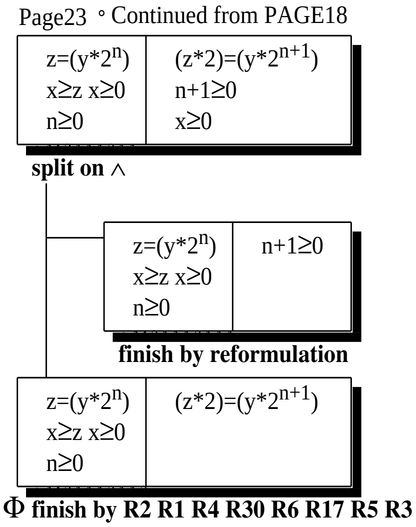
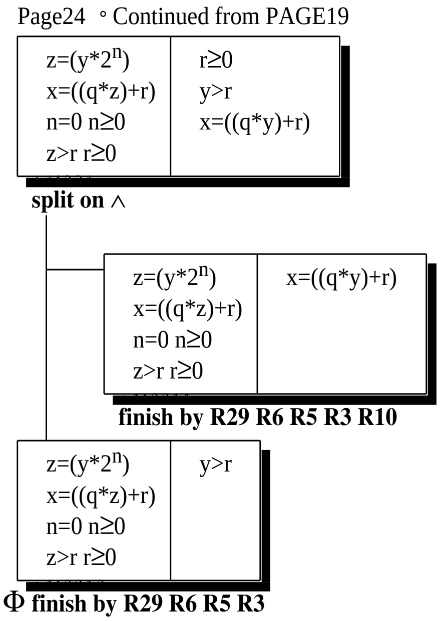


Figure 10: Continuation of the proof for the quotient-remainder procedure, showing the use of Hoare rules to eliminate the second while statement.



Rules

R1	$1 \equiv \text{add1}(0)$
R2	$2 \equiv \text{add1}(\text{add1}(0))$
R3	$x+0 \equiv x$
R4	$x+\text{add1}(y) \equiv \text{add1}(x+y)$
R5	$x*0 \equiv 0$
R6	$x*\text{add1}(y) \equiv x+(x*y)$
R17	$x*(y+z) \equiv (x*y)+(x*z)$
R30	$x^{\text{add1}(y)} \equiv x*x^y$



Rules

R3	$x+0 \equiv x$
R5	$x*0 \equiv 0$
R6	$x*\text{add1}(y) \equiv x+(x*y)$
R10	$x>y \equiv y<x$
R29	$x^0 \equiv \text{add1}(0)$

Figure 11: Proofs of two of the first-order formulas (verification conditions) generated by application of Hoare inference rules on the quotient-remainder procedure (from two pages of actual output). The rules shown in the Rule table are axioms given to define integer operations (+, *, add1, power) or lemmas about these operations. At the time the program proof was carried out the lemmas had already been proved automatically by Tecton.

6 Comparison with other proof approaches

As should be evident from the discussion in this paper, significant features of the Tecton proof management system include (i) its powerful inference engine with built-in-heuristics for automating proofs by induction, rewriting (in particular contextual rewriting), automatic case analysis, and linear arithmetic decision procedure, (ii) representation and structuring of proof attempts using proof forests, and (iii) proof visualization using tabular formats for formulas, graphical depictions of proofs, and cross-referencing via hypertext links. In comparing the Tecton Proof System with other approaches, we divide the discussion into three parts corresponding to these features. Systems and environments such as Mural, Raise, and B-toolkit provide proof checkers and proof management systems that are tightly integrated with their specification language. For this reason it is somewhat difficult to discuss their reasoning tools in isolation. However, we will refrain from comparing the Tecton specification language with other specification languages and systems, as such a comparison is outside the scope of this paper. An interested reader may consult [19].

6.1 Proof construction and representation

6.1.1 Basic proof management issues

The approach to proof structure and management used in Tecton is based in part on the notion of proof forests supported in the Affirm system [9]. Other systems that are similar in their proof support to Affirm include the proof assistants of the Mural [30], EVES [10], and PVS [35]. Mural allows users to introduce their own inference rules that can be used in a proof even if the inference rule is not yet proved. As in Affirm and Tecton, proofs can be attempted top-down, bottom-up or in a mixed strategy. Mural keeps track of the use of unproved inference rules using a dependency mechanism to avoid any circular reasoning. It allows multiple proof attempts in a proof forest by maintaining multiple copies; in contrast, Tecton maintains only one copy and instead provides for alternative inference steps below any goal. The PVS and EVES also maintain a tree of goals and subgoals, but without provision for alternative inferences.

6.1.2 Goals, subgoals, and tactics in LCF-style systems

Some form of top-down transformation of goals into subgoals is provided by several other natural deduction style provers. LCF [14, 36] and several LCF-influenced systems, including Nuprl [7], HOL [15], Isabelle [38], and Lego [29], use *tactics*, which are programs for reducing a goal to a list of subgoals. From a goal g, which is a formula, and a list of formulas regarded as assumptions, a tactic produces a list of subgoals $g_1, \ldots, g_n$ and a *justification.* The latter is a function that, when applied to a list of theorems, produces a theorem that "achieves" the original goal g (see [15, Section 7.1] for the precise sense in which a theorem achieves a goal). The theorems to which the justification is applied are those theorems that achieve the subgoals $g_1, \ldots, g_n$. The justification cannot be applied until those theorems are available, which is after each of the subgoals has been proved. Finding a proof is thus a top-down process that is followed by an automatic bottom-up construction, by means of the justifications, of the proof itself, showing that the original formula is achieved by a theorem. The bottom-up phase, in combination with the type-security of the host language (theorems are values of a type whose only constructors are axioms and primitive rules of inference), ensures that soundness is preserved, i.e., it is not possible through the use of tactics to prove an invalid theorem (assuming that the axioms and primitive rules of inference are sound). Thus these systems are safe in allowing users to extend the prover by writing their own tactics. Tecton currently does not provide such an extension facility, but its built-in inference mechanisms are much more powerful than those found in any of the LCF-style systems (see below).

In LCF-style systems, a goal stack is built up in the course of searching for a proof, but the stack is not preserved; when the proof is complete, there is no data structure that preserves a record of the steps taken in searching for the proof. To repeat the proof one would need to reenter the commands used in manipulating the goal stack. More importantly, one would like to preserve the pattern of use of tactics so that it can be repeated when attempting proofs of other formulas. Typically the system user must rely on the transcript of the steps taken in finding the proof as a guide to the construction of a *single tactic* that, when applied to the original goal, achieves the goal.

When a tactic fails there is very little feedback about where it is failing, and thus little aid in determining how an existing tactic should be modified to make it more generally useful. If one could see the subgoals that are being produced by a tactic when applied to a particular goal (without having to break up the tactic into smaller pieces and apply each piece separately), one could gain more insight into possible ways of restructuring the tactic. The proof forest and proof display facilities of Tecton are designed to provide and preserve a more complete picture of intermediate stages of proofs.

The rationale sometimes given for not keeping a record of inference steps is that it would grow too large; e.g., "an Isabelle proof typically involves hundreds of primitive inferences and would be unintelligible if displayed" [38]. Tecton proofs also typically involve hundreds of primitive inferences, but inference nodes in Tecton proof forests organize large groups of primitive inferences into meaningful macro inference steps.

6.1.3 Automatic versus interactive proof

The B-toolkit [3], a collection of software tools designed to support development of software systems using the *B-method*, includes two theorem provers, called the *auto* prover and *inter* prover. The *auto* prover uses built-in tactics and a standard library of definitions, theorems and lemmas about well-known mathematical structures to discharge proof obligations. Proofs that do not succeed are passed to the *inter* prover, which allows a user to investigate why the *auto* prover has failed to discharge an obligation and to interactively attempt a proof using tactics and additional lemmas that may not be in the *auto* prover's standard library. User-provided tactics augment the built-in tactics of the *auto* prover and are tried if the built-in tactics do not succeed. In Tecton, on the other hand, the automatic and interactive phases of proof development are intertwined. Tecton relies heavily on its built-in heuristics and decision procedures for generating proofs automatically. When attempts at automatic proof fail, the user can select a formula node anywhere in the proof forest and continue the proof interactively with commands that force specific inference steps to be constructed as alternatives to the steps of failed proofs.

6.1.4 Formal versus informal justifications

Tecton and the other systems discussed up to this point emphasize formal proof; the RAISE (Rigorous Approach to Industrial Software Engineering) system, on the other hand, places more emphasis on recording dependencies and design decisions, without worrying too much about whether such design decisions can be rigorously justified. The RAISE language is a wide spectrum language for "very abstract, initial specifications and also for concrete development of initial specifications that can be easily (or even automatically) translated into a programming language" [11]. The RAISE Toolset [12] includes a *justification editor* for simplifying expressions and expanding goals into subgoals, to be used for interactively developing justifications for correctness of a specification. A justification is an argument why a justification condition is believed to hold, where a justification condition may be a formal condition such as a formula, or a confidence condition which is considered a warning. A justification may be a formal proof developed using proof rules, definitions and theorems in a theory, or it can be completely informal or a mixture of both. This notion of justification is broader than formal proof and and in some cases may be a more suitable basis for recording software design decisions; we plan to examine this question further as we attempt to integrate the Tecton specification language with the proof system.

6.2 Proof visualization

The visualization techniques used in the Tecton system make an attempt to present proofs to a user in a more comprehensible form. This is needed if proofs are to be reused, restructured, or generalized. Tecton's page-by-page layout of proofs also allows them to be included directly in reports and documents (see, for instance, the example proofs in Section 5).

HOL, Isabelle, Lego, and related proof checkers do not provide any graphical depictions or hypertext mechanisms for visualizing the structure of proofs. Once a proof is generated by a series of commands, a single tactic can be constructed (manually) to record the inference steps, but not the intermediate subgoals, of the proof. As previously noted, however, such large tactics are not by themselves adequate for human understanding; to show the details of the proof, tactics have to be broken back down into a sequence of smaller tactics or tactics that are then replayed to see the subgoals they produce.

Théry [40] discusses principles for providing graphical interfaces to proof checkers in general, and in particular to the HOL system. However, we believe such approaches are not likely to succeed unless they are based on some well-chosen record of proof structure such as the Tecton notion of proof forest; without that, proofs are bound to be either too detailed or not detailed enough for easy comprehension, no matter what graphical form is used.

6.3 Inference rules and mechanisms

The LCF, HOL, Isabelle, Lego and other higher-order proof checkers lack decision procedures and fully automated rewriting. As a consequence, the user must often carry out proofs in excruciating detail. The inference engine of PVS does contain a collection of decision procedures for ground theories [35], but rewriting is not fully automated.

The rewriting capability in HOL is adapted from the LCF implementation of rewriting [37]. There is no automatic application of rewrite rules; each equational theorem to be used as a rewrite rule must be named explicitly.[4] In Tecton, by contrast, rules are applied automatically and there may be hundreds or thousands of rewrites taking place in a proof without the user's direct involvement.

Most higher-order proof checkers such as HOL, Isabelle, Lego, PVS, etc., support a highly expressive meta-logic which can be used to encode many different logics and formal notations. PVS, for instance, supports a sophisticated type definition mechanism; checking whether a value belongs to a particular type may involve performing a complex proof. Isabelle is a generic theorem prover in which logics can be introduced by specifying their syntax and rules of inference, and

[4]A recent version of Isabelle (Isabelle-93) has reported an implementation of a simplifier for rewriting.

proof procedures can be expressed using tactics. Proofs in Isabelle are performed by a generalization of resolution using higher-order unification, which is undecidable. Lego [29] is another interactive proof checker based on type theory in the natural deduction style, but Lego does not seem to support tactics. An interesting aspect of Lego is that it has been used to encode, among other formal systems, specification languages *Z* and *VDM*. It is however unclear to us how useful and effective such proof checkers can be in reasoning about specifications written in *Z* or *VDM* or programs written in a programming language. In general there is tension between the expressiveness of a underlying meta-logic supported in a proof checker and the efficiency and automation of inference mechanisms provided to reason about meta-linguistic constructs.

Our approach in developing the Tecton proof system has been to provide powerful inference mechanisms for equations and conditional equations with the hope that most theorems can be automatically proved, so that the user can focus on interesting, nontrivial aspects of specification design and verification, and expending proof efforts on nonobvious theorems. In this process, the user is expected to gain more insight as well as confidence into the formal model being built using specifications and programs. In contrast to proof checkers which rely heavily on the user to define tactics to perform large inference steps, our philosophy in developing Tecton's inference engine has been to build in algorithms that are known to be useful and perform well. For these reasons, we have not tried to integrate higher-order constructs in Tecton.

Tecton's inference engine is similar in its objectives and style to Boyer and Moore's theorem prover [4], in that both theorem provers provide rewriting, heuristics for proofs by induction, and a decision procedure for linear arithmetic. The methodology for generating induction schemes in the Tecton system is closely related to the method in Boyer and Moore's prover, with the main distinction being that definitions are given as Lisp-like functions in Boyer and Moore's logic, whereas in Tecton they are given as sets of terminating rewrite rules.

The inference engine of the Tecton system is also related to the Larch prover [17] in that both theorem provers accept function definitions as terminating rewrite rules. The Larch prover provides a user access to a variety of inference methods, but many appear to require more interaction than Tecton. In particular, the principle method for performing induction supported in the Larch prover is generator induction derived from generators of a data type, which does not automatically find a proof as often as the cover set induction method.

7 Conclusion

We have described some of the novel features of the Tecton Proof System in the areas of proof representation, visualization, and automation. The chosen proof representation, as a forest of proof trees, is designed to be structurally relatively simple while still allowing considerable aid in reducing the size of proofs and flexibility in maintaining multiple proof attempts. The proof visualization techniques were chosen with the goal of expressing complex proofs about computations in ways that make the proofs more easily comprehensible. By simplifying and organizing the way programs, assertions, and proofs are expressed—using tables, graphics, and a combination of cross-indexing and hypertext techniques—Tecton does, we believe, greatly improve the readability of specifications and proofs. In the area of proof automation, we have described the main inference mechanisms including Hoare proof rules, reduction, case analysis and splitting, induction, generalization, and a linear arithmetic decision procedure that is well integrated into the other mechanisms.

The Tecton Proof System has been used to prove properties of simple (but efficient) programs on integers (the quotient-remainder program in Section 5 is such an example), equational programs for sorting and searching, and simple programs using abstract data types.

The Tecton Proof System is an experimental system and is only one component of the broader Tecton framework we envision. The following are the other components we are working to design, develop, and integrate:

- The *Tecton specification language* [19], a high level language for describing and using abstract concepts in formal software development and hardware design.

- A *proof command language* for interactively guiding the construction of large, complex proofs, allowing the user to keep control of proof development while making effective use of the automated deduction capabilities of the system.

- A *program modeling language* [31] capable of modeling crucial features in production programming languages, such as Ada and C++, for building libraries of generic software components: encapsulation, templates/generics, and class inheritance. Tecton currently supports only the simple kind of program modeling language used in most of the literature on Hoare's axiomatic formalism.

- *Data base facilities* for storing, indexing, and retrieving large collections of specifications, designs, programs, axioms, theorems, and proofs.

Integration of the Tecton language with the proof system is particularly crucial to our goal of supporting specification and verification of generic software and hardware design components. The Tecton language provides *definition, abbreviation, extension*, and *lemma* constructs, which have general mathematical descriptive power, plus a computation-specific *realization* construct. The semantics, which is denotational, includes specification of the requirements ("legality conditions") that must be met when using each construct. The syntax and semantics are such that a corresponding proof theory requires only first order and inductive proof methods, rather than general higher order techniques as required in some frameworks. The current design of the language and some of the main proof issues are described in [19], which also contains a substantial example of its use in hardware design—a behavioral and structural description of a carry-lookahead adder circuit, with the circuit realization given in terms of a generic parallel-prefix circuit. Hierarchies of concepts useful for symbolic algebra systems as well as for linear data structures are discussed in [20]. An example of specification and verification of a generic software component—an efficient partitioning algorithm—may be found in [25].

We have also implemented and are currently experimenting with a simple proof command language. The command language supports application of primitive inference steps such as *reduction*, *induction,* including using an induction scheme specified by a user, *generalize,* and *case analysis,* as well as commands for displaying a data base of definitions, theorems and lemmas as rewrite rules, and commands for browsing through a proof.

Acknowledgments. Raj Agarwal, Michael Seager, and Changqing Wang have assisted in the implementation of Tecton, particularly of the program parser, Hoare formula generator, and program table displayer. Chuck Kacmar gave early advice and encouragement on the use of hypertext techniques. We thank Mahadevan Subramaniam for assistance with the examples in Section 5 and useful comments on an earlier draft of this paper. The referees also made many useful suggestions for improvement of the paper.

References

[1] R.M. Akscyn, D.L. McCracken, E.A. Yoder. “KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations,” *Comm. ACM* 31(7): pp. 820–835 (July 1988).

[2] Avron Barr and Edward A. Feigenbaum, *The Handbook of artificial intelligence*, W. Kaufman, 1981.

[3] *B-Toolkit: Specification and Design Assistant, User's Manual,* BP International Ltd., 1992.

[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

[5] R.S. Boyer and J S. Moore, "Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic," *Machine Intelligence* 11 (1988) 83–157.

[6] J. Conklin, "Hypertext: An Introduction and Survey," *IEEE Computer* 2(9): pp. 17–41 (1987).

[7] R. Constable, *et al.*, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, 1986.

[8] N. Dershowitz, "Termination of rewriting", *J. Symbolic Computation* 3: pp 69–116 (1987).

[9] R.W. Erickson and D.R. Musser, "The AFFIRM theorem prover: proof forests and management of large proofs," *5th Conference on Automated Deduction, Lecture Notes in Computer Science*, Vol. 87, Springer-Verlag, New York, 1980.

[10] D. Craigen, *et al*, "Eves system description," *Proc. Automated Deduction – CADE-11,* LNAI 607, Springer Verlag, 1992, 766–775.

[11] C. George, "The RAISE specification language: a tutorial," *Proc. of VDM'91, Formal Software Development Methods,* LNCS 552, Springer Verlag, Vol. 2, 1991, 239–319.

[12] The CRI Raise Tools Group, "The Raise Toolset," *Proc. of VDM '91, Formal Software Development Methods,* LNCS 551, Springer Verlag, Vol. 1, 1991, 689–690.

[13] J. A. Goguen, T. Winker, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, "Introducing OBJ," to appear in J.A. Goguen, D. Coleman, and R. Gallimore (editors). *Applications of Algebraic Specification using OBJ*, Cambridge University Press, 1992.

[14] M. J. Gordon, R. Milner, and C. P. Wadsworth, *Edinburgh LCF*, LNCS 78, Springer Verlag, 1979.

[15] M. J. Gordon, "A Proof Generating System for Higher Order Logic," *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subramanhanyam, eds., Kluwer, G. Birtwistle and P.A. Subramanhanyam, eds., Kluwer, 1987.

[16] J. V. Guttag and J. J. Horning, "Report on the Larch Shared Language," *Sci. Comput. Program.*, vol. 6, no. 2, pp. 103–134, Mar. 1986.

[17] J. Guttag, et. al., *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science, Springer Verlag, 1993.

[18] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM* 12 (10) (1969) 576–583.

[19] D. Kapur and D.R. Musser, *Tecton: a framework for specifying and verifying generic system components*, Rensselaer Polytechnic Institute Computer Science Technical Report 92-20, July, 1992. Invited talk at *TPCD Conf. 1992 (Theorem Provers in Circuit Design)*, University of Nijmegen, The Netherlands, June 22-24, 1992.

[20] D. Kapur and D.R. Musser, *Examples of Tecton concept descriptions*, in preparation.

[21] D. Kapur, D.R. Musser and X. Nie, "The Tecton Proof System," chapter in *Formal Methods in Databases and Software Engineering,* Alagar, Lakshmanan, Sadri, eds., *Workshop in Computing Series*, Springer Verlag (1993) 54–79.

[22] D. Kapur, D.R. Musser and X. Nie, *The Tecton Proof System: Introduction and User's Guide*, Institute for Programming and Logic, State University of New York (1991).

[23] D. Kapur, D.R. Musser, and A.A. Stepanov, "Operators and algebraic structures," *Proc. of Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire (1981).

[24] D. Kapur, D.R. Musser, and A. A. Stepanov, "Tecton: a language for manipulating generic objects," *Proc. of Program Specification Workshop, University of Aarhus, Denmark*, August 1981, *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 134 (1982).

[25] D. Kapur, D.R. Musser, and C. Wang, *Specification and verification of a generic partitioning algorithm,* in preparation.

[26] D. Kapur and X. Nie, *Reasoning about numbers in Tecton,* Department of Computer Science, State University of New York, Albany, NY 12222, March 1994.

[27] D. Kapur and H. Zhang, "An overview of Rewrite Rule Laboratory (RRL)," to appear in a special issue of *Computers in Math. with Applications,* 1994. Earlier descriptions appeared in CADE-88 and RTA-89.

[28] R.A. Kemmerer, "Verification Assessment Study Final Report", Vols. 1-5, National Computer Security Center, Fort George G. Meade, Maryland.

[29] Z. Luo, and R. Pollack, *Lego Proof Development System: User's Manual,* Department of Computer Science, Univ. of Edinburgh, May 1992.

[30] R. Moore, "Machine support for formal methods: an overview of Mural," Unpublished Manuscript, Dept. of Computer Science, University of Manchester, Manchester, UK, 1992.

[31] D.R. Musser, *Elements of a Pragmatic Approach to Program Verification*, Rensselaer Polytechnic Institute Computer Science Department TR 89–24 (1989).

[32] D.R. Musser and A.A. Stepanov, *The Ada Generic Library: Linear List Processing Packages*, Springer-Verlag Compass Series (1989).

[33] G. Nelson and D.C. Oppen, "Simplification by Cooperating Decision Procedures," *ACM Transactions on Programming Languages and Systems* 1 (2) (1979) 245–257.

[34] T. Nipkow and L.C. Paulson, "Isabelle-91," *Proc. Automated Deduction – CADE-11,* LNAI 607, Springer Verlag, 1992, 673–677.

[35] S. Owre,, J.M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," *Proc. Automated Deduction – CADE-11,* Saratoga Springs, LNAI 607, Springer Verlag, 1992, 748–752.

[36] L.C. Paulson, *Interactive theorem proving with Cambridge LCF: a user's manual*, Technical Report 80, Computer Laboratory, University of Cambridge, Cambridge, England, 1985.

[37] L.C. Paulson, "A Higher-Order Implementation of Rewriting," *Science of Computer Programming*, 1983.

[38] L.C. Paulson, Introduction to Isabelle, Computer Laboratory, University of Cambridge, England, 1993.

[39] R.E. Shostak, "A Practical Decision Procedure for Arithmetic with Function Symbols," *Journal of ACM* 26(2) (1979) 351–360.

[40] L. Théry, Y. Bertot, and G. Kahn, "Real theorem provers deserve real user-interfaces," University of Cambridge report, May, 1992.

[41] H. Zhang, D. Kapur, and M.S. Krishnamoorthy, "A Mechanizable Induction Principle for Equational Specifications," Proc. of *Ninth International Conference on Automated Deduction (CADE-9)*, Argonne, IL (1988).

[42] H. Zhang, "Implementing contextual rewriting," in: Proc. *Third International Workshop on Conditional Term Rewriting Systems,* J. L. Remy and M. Rusinowitch (eds.), Lecture Notes in Computer Science, Vol. 656, Springer-Verlag, Berlin, 1992, pp. 363–377.